# Compiling Curried Functional Languages to .NET

---

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
by
Jason Smith

---

University of Canterbury
2004

To God, Dad, Mum, Talei and Laura

# Abstract

Recent trends in programming language implementation are moving more and more towards "managed" runtime environments. These offer many benefits, including static and dynamic type checking, security, profiling, bounds checking and garbage collection. The Common Language Infrastructure (CLI) is Microsoft's attempt to define a managed runtime environment. However, since it was designed with more mainstream languages in mind, including $C^\sharp$ and C++, CLI proves restrictive when compiling functional languages. More specifically, for compilers such as GHC, which compiles Haskell, the CLI provides little support for lazy evaluation, currying (partial applications) and static type checking. The CLI does not provide any way of representing a computation in an evaluated and non-evaluated form. It does not allow functions to directly manipulate the runtime stack, and it restricts static typing in various forms; including subsumption over function types.

In this thesis, we describe a new compilation method that removes the need for runtime argument checks. Runtime argument checking is required to ensure proper reduction semantics in situations where the arity of a function may be statically unknown. We introduce a new set of annotations, called *operational types*, which provide an abstract model of reduction. From these operational annotations we can construct a transformation, called *lambda doping*, which saturates all partial applications, removing the need for runtime argument checks. This enables the CLI to support an eager, curried, higher order functional language.

We also describe a type inference algorithm that infers universally quantified types with implicit widening coercions. We show that we can easily include the generation of operational typings from type inference. We restrict type inference to simple extensions of the usual unification algorithm, given by Hindley-Milner, so that inference is immediately applicable to most commercial functional languages. We also develop a set of transformations which demonstrate that for the most part inferred types can be readily mapped to the CLI.

Finally, we develop a practical implementation of a higher-order, curried eager functional language, called Mondrian.

# Acknowledgments

Many people have contributed in so many ways over this odyssey.

Firstly, thank you to Dr. Nigel Perry, my supervisor. Thank you for persevering with me when you no doubt wondered if I would indeed get anything done. Thank you for the opportunities you have presented to me along the way, the ones I took up as well as the ones I missed.

Secondly, thank you Dr. Wolfgang Kreutzer, my secondary supervisor, for your great suggestions on style and content.

I would like to give my deepest thanks to my family, Dad, Mum and my little sister. I can't imagine the countless number of times I rang home depressed about my lack of progress, only to receive never-ending encouragement. Thank you for all the curries and the roasts you sent via overnight delivery, even though my "great cooking skills" secret was eventually discovered. Thank you, Dad, for all the pep talks and "I don't want to die knowing u haven't finished your MSc yet son" statements. Thank you, Mum, for the constant prayer requests and "did u go to mass?" questions. Thank you, Sis, for being around and joking with me whenever I needed it! Thank you for all the time you spent in the last weeks checking my atrocious grammar. If this thesis is comprehensible, it is in no small part attributable to you. Finally, all those banana cakes made me a popular guy at the office.

In no particular order:

- **Laura** : I think you were almost as much a part of my life over the past few years as this MSc. Thank you for always being around when I needed to talk about anything and everything, and making me laugh whenever I needed cheering. Lastly, just for you, I have this quote I picked up on my Internet travels:

  > Her not my excellent grammar to criticize I told. – Angry Yoda

  I owe you a new dinky winky key and a big dinner in Cologne for all

the hours of proofing you did! A thank you also goes out to Kristina, since no thank you would be complete without the other half of the Xendom duo.

- **Josh** - Thanks for all the NRG sessions we shamelessly attended, the many supermarket runs and your endless winning streaks at Generals, and for all the help you offered whenever you could.

- **Behshid** - Thanks for all the extra pounds I gained with all the delicious cakes you saved for me, for the great laughs and good times we shared, including the numerous weird conversations.

- **Odette** - I forget all those late night "study" sessions, the 1 am tangos in the crypt and the movies we watched at 3 am.

- **Ioane** - Thanks for helping me lose said pounds with our 10km runs in the woods. Your're a one of a kind dude and probably the only person with whom I'll ever run another half marathon.

- **Aun** - Thanks for the many Urdu translations and the many long lonely vigils as we both worked on our theses.

- **Mike C** - Cheers Mikolstovosky for all the late night "ping" sessions on MSN while you were working late in the lab.

- **Binh** - Thanks for the great times we spent on the hockey rink, the "golf" lessons and the many laughs.

- **Mike J S** - Thanks, latex lad, for all the interesting random information and latex formatting help. You were an inspiration for Josh and I to finish.

- **Nadine and Rahim** - You guys have both contributed to making this experience unique.

Table of Contents

# List of Figures

# List of Tables

# Chapter I

# Introduction

We are interested in looking at several aspects of functional language compilation to the ECMA/ISO CLI. Our discussion focuses on three aspects, runtime support for, lazy evaluation and partial applications and static type inference. We will be discussing this in the context of the Mondrian language.

## *1.1 The CLR*

The CLI and $C^\sharp$ are defined by the ECMA/ISO standards 334 and 335 respectively[1]. The CLI defines a common language type system over the intermediate language IL. This allows any language targeting the CLI to interact in a type safe manner with any other language also targeting the CLI. The CLI provides a number of common features, including value types, boxing/unboxing, reference types and subsumption; both constrained and unconstrained. Type checking is enforced by the CTS, or Common Type System. It checks that types are manipulated in a type safe manner. A program is deemed type safe if it is statically typeable by the CTS. Static typing provides a degree of safety, in that well-typed programs do not "go wrong". We differentiate between two versions of the CLI, namely the $\text{CLR}_{\leq}$ and $\text{CLR}_{\leq,\forall}$. The $\text{CLR}_{\leq}$ is the initial version of the CLR. Among other things the CTS for the $\text{CLR}_{\leq}$ provides value types, reference types, name equivalence, inheritance using subtype polymorphism and encapsulation. $\text{CLR}_{\leq,\forall}$ extends this by providing support for generics. Generics provides a way of parameterising a computation. Unlike subtype polymorphism, which is polymorphic over a subtype hierarchy (established by inheritance), generics allows us to parameterise over unrelated types.

[1] Found at www.ecma-international.org/publications/standards/Ecma-335.htm

## *1.2 Issues*

We briefly look at three main issues for the compilation of functional languages to the CLI, which are: lazy evaluation, partial applications and static typing. We wish to look for solutions to these problems that do not require modification to the CLI in any fashion.

### *1.2.1 Lazy Evaluation*

Lazy evaluation allows an expression's evaluation to be delayed, by suspending the computation until it is required. This is an especially helpful feature when we want to limit the amount of memory usage and unneeded computation.

Lazy evaluation presents a problem when compiling to the CLI because there is currently no natural way of representing delayed computation because of strong typing. A value can be either unevaluated or evaluated and we have to "switch" on this behaviour to allow different calling semantics. A possible solution is to use an inheritance based mechanism to express the idea that a value is either of type $\tau$ or of some form of "unevaluated-$\tau$". While this works for reference types, in its current form it is not appropriate for value types as the CLI restricts the ability to inherit from value types by sealing them. Thus, this solution would require modifications to the CLI.

### *1.2.2 Partial Applications*

We define a partial application as the application of a function of arity $a$ to some number $n$ of arguments such that $n > 0$ and $n < a$. Partial applications are allowed in languages that support currying. A curried language treats application as the repeated application of a function to single arguments. We call a program unsaturated if it executes with partial applications. It is unsaturated in the sense that not every function in every application is applied to all its arguments. This is as opposed to a program that is saturated in which in every application every function is applied to all its arguments.

At runtime, support for partial applications requires a form of runtime argument checking. A runtime argument check checks that there are suffi-

cient arguments on the stack to make the function call. If there are not, then a form of closure is constructed, encapsulating the supplied arguments and the partially applied expression. This expression is finally evaluated when the remaining or pending arguments are supplied.

A runtime argument check stipulates the ability to "mark" the stack in some fashion. Unfortunately the CLI prevents programs from directly manipulating the stack, as does the Java Virtual Machine. Doing so could lead to malformed execution and security holes. This problem is not restricted to the CLI. Another portable intermediate assembly language, C–, (Peyton Jones et al. 1999) also does not allow direct stack manipulation, including not allowing arguments to be kept on the stack past the lifetime of the call. C– provides detailed control over stack frame layout, provides support for tail calls, register allocation for procedure invocation and multiple return values. In spite of this there is currently no elegant way of extending the C– intermediate assembly language to meet this requirement. This situation forces compilers such as GHC,(Marlow & Jones 1997) to completely reimplement many low-level features, such as stack management, in C. The problem is also shared by a number of other languages, such as $F^\sharp$, (Syme 2002) and *Fleet* (Faxn 1996*b*).

Obviously abstracting the system stack, as does GHC, is not a desirable strategy. If we can saturate all function calls in some way, then we will no longer require the ability to manipulate the system stack. We will look at this problem in more detail later.

### *1.2.3 Type Inference*

The enforcement of the type discipline takes place either statically or dynamically. A static type enforcement policy mandates the checking of programs before the program is executed, to make sure it is *well-typed*. A programmer in a dynamically typed language does not consider how the data he uses, be it integers, reals or strings, are handled by functions acting on a subset of this data. In contrast a static language enforces a typing discipline, requiring functions that act on different types of data to take disjoint unions. This means that the function that takes a disjoint union of *integer* and *float* is

different from a function that takes just a float. A type discipline allows us to classify programs as being well-typed, which are usually held to two criteria in the literature; i.e. no well-typed program has any applications that will give run-time errors, and the property of being well-typed must be *decidable* for arbitrary programs.

Static type checking is preferable because it gives us a weak form of verification, assigns types (reducing code obfuscation) and provides a number of ways of performing optimisations/transformations. This last point being especially critical when compiling to a restrictive runtime environment, as in our case. A dynamic typing discipline in contrast accepts all programs. Undefined program execution is detected during program execution.

The CLI provides both static and dynamic type checking. Coercions from subtypes to supertypes can be checked statically. These coercions are implicit in the sense that they will happen automatically. Coercions from *Object*, CLI's $\top$ type, to a subtype are narrowing coercions and have to be included explicitly and are checked dynamically.

### *1.3 Research Objectives*

Now that we are in a position to frame our research objectives, we decide to focus on two aspects:

1. Develop a type inference algorithm that will be at least as expressive as that inferred by a Hindley-Milner type scheme. By this we mean it will type at least as many programs as Hindley-Milner.

2. Remove the need to support runtime argument checking. This gives us a way of compiling a functional language supporting currying *efficiently* to the CLI.

We approach these issues in the context of the Mondrian language. Mondrian is a higher order curried functional language which can be viewed as a subset of the Haskell98 [2] standard with syntax to give Mondrian programs a

[2] available from http://www.haskell.org/definition/

more imperative look. Mondrian started as a research project at Utrecht University, (Meijer & Claessen 1997) and (Meijer et al. 2001), and was designed primarily as a teaching aid. Mondrian is a *pure* functional language and as such doesn't enjoy some of the same benefits of interacting with the .NET runtime, as impure languages such as SML.NET[3] (Kennedy et al. 2003) and $F^\sharp$ [4] (Syme 2001). Mondrian compiles to the Common Language Infrastructure (CLI). and was originally designed to rely on type inference. However its implementation relied on the $C^\sharp$/CLI to do most of the typing and did not have its own type checker. Mondrian relied on the runtime widening coercions provided by the CLI and was dynamically typed as runtime exceptions could occur. Such errors are in the same nature as division by zero, incomplete case distinction, and other runtime errors caused by partial functions in Haskell.

We take a closer look at two issues: runtime argument checking and type inference.

### *1.3.1 Runtime Argument Checking*

We take a closer look at why runtime argument checking is required, which may suggest a solution. The stack can manipulated using two generally accepted methods, via the *push/enter* model or the *enter/apply* model (Peyton Jones 1992).

- Eval/Apply - Popularised in the early 80's by compilers based on the Lisp strategy of: evaluate the function, evaluate the argument and apply the function value to the argument. It is traditionally used by strict languages (e.g. Lisp, Hope, and SML).

- Push/Enter - Based on the graph-reduction model, push the argument on the evaluation stack, and tail-call (or enter) the function. This is usually used by the G-machine (Augustsson 1984), TIM (Fairbairn & Fradet 1987), and the STG machine (Peyton Jones 1992).

[3] available from http://www.cl.cam.ac.uk/Research/TSG/SMLNET/download.htm

[4] available from http://research.microsoft.com/projects/ilx/fsharp-release.aspx

The differences between the two strategies are apparent when we are considering an *unknown* function call. An unknown function is one for which we do not know its arity statically, which is usually the case when considering a polymorphic function. This is the opposite of a *known* function call where we know the arity of the function call statically.

Consider the sample below, taken from (Marlow & Jones 2004). It illustrates the use of an unknown function:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith k [] []          = [];
zipWith k (x:xs) (y:ys) = k x y : zipWith xs ys;
```

Listing 1.1: Polymorphic function

In this case `k` is the unknown function. We do not know the arity `k` is instantiated to. We do specify in the "contract" of the type signature that we are going to apply this function to two arguments. This means that in the function call `(k x y)` we can't simply apply the function f to the two arguments, because the exact arity of f is unknown *within the definition* of `zipWith`. This delineation between caller and callee is important, as the caller always knows what it is passing to the `zipWith` function. If we were evaluating the function using eval/apply, the function `k` would be evaluated to its canonical form and its arity would be extracted at runtime. The function always knows how many arguments it needs. Depending on how many arguments are available, we either complete the call or generate a partial application (pap). If there are any arguments remaining on the stack, these are presumably *consumed* by the function returned from the initial application. In the eval/apply model the remaining arguments are used to create the *call continuation*, the rest of the original call, when the original application returns.

Push/enter, in contrast, pushes the arguments `x` and `y` onto the stack and *enters* the code for `k`. The function, knows how many arguments it needs and checks the stack for the appropriate number producing a partial application if appropriate. The function "consumes" as many arguments as it needs, leaving the remaining arguments on the stack. There is no "call continuation"; functions push and pop directly from the stack.

Marlow & Jones (Marlow & Jones 2004) summarise the differences between the two models in how they place responsibility for arity-matching:

- Push/Enter - the *callee*, which statically knows its own arity, examines the stack to determine how many arguments it has been passed and where they are stored.

- Eval/Apply - the *caller*, which statically knows what the arguments are, examines the function closure, finds its arity and makes a call to the function.

Where the number of arguments in both cases is less than that required by the function, a closure is built within the callee's body of code storing the arguments applied at that point which is called a *partial application*. For example, in Haskell:

```
f   = x -> y -> z -> x + y + z;
f2  = map f [1, 2, 3]
map = g -> l -> case l of
                  {  []    : []
                  ;  x::xs : (g x) :: map f xs;
                  }
```

The function `map` expects a function with only 1 argument, yet we provide `f` that expects 3. The application `f x` generates a partial application.

We remarked that while the function bound to the polymorphic argument `g` is unknown, it's interface to the outside world is known. This motivates a interesting observation. Is it possible to guarantee that the function receives what it wants at all times? If we can guarantee that the function `zipWith` will always receive a function of arity 2, then it no longer needs to perform a runtime arity check. Solving this means bringing these two pieces of information, the arity of the function and the number of arguments, together during compile time.

Aside from the location of arity checks, both models provide operational advantages and disadvantages. Push/enter is inherently suited towards compiling a curried language, as the call proceeds by pushing the arguments `x` and `y` onto the stack and then entering the code for `k`. The function `k` can

consume as many arguments as required and leave the remaining arguments on the stack for future calls. Unfortunately this also makes it completely inappropriate for the CLI as we mentioned. We cannot produce code that allows the function to manipulate the system stack. We can solve this by abstracting the system stack, as is done by GHC. This constituted the initial implementation of Mondrian, called the Mondrian Exceptional Machine or ($u$-MEM).

We do not have these same problems in eval/apply since we evaluate the function first, and then apply it to the arguments. Any remaining arguments are consumed by the "rest" of the evaluation. No arguments remain on the stack past the lifetime of the call and all pending arguments are contained within a structure that allows us to record any additional information about the types of the arguments. The CLI does support an inefficient form of variable length application using varargs, but these are very inefficient are not considered. The explicit application to some number of arguments makes the eval/apply model a better choice for compilation to the CLI, as it fits the semantics of the CLI's function application.

### *1.3.2 Current Implementations*

Now that we have identified the problem we need to examine how compilers have so far solved the problem. A large body of literature exists on the internals of GHC compiler, which we do not reiterate. Full operational semantics for GHC can be found in (Marlow & Jones 2004). Our examination of the compilation model for GHC concerns how it handles testing for partial applications.

#### *Compilation with Runtime Argument Checking*

In push/enter, GHC uses a special register to mark the stack. The mark delimits the "current" stack, effectively creating a new stack whenever a suspension is evaluated. It stops the suspension from consuming values from outside its scope. The argument satisfaction check is the result of subtracting the mark register from the stack frame pointer. GHC generates functions with two entry points, called the fast and slow entry points. This is because

the location of the arity check is within the function. The fast entry point is used for known calls, expecting its arguments in registers (plus some on the stack, if there are too many to fit in registers). The slow entry point expects all its arguments on the stack and begins by performing the argument satisfaction check. If it succeeds, the slow entry point loads the arguments from the stack into registers and jumps, or rather "falls" through, to the fast entry point. Otherwise, assuming that we have the correct number of arguments on the stack, the slow entry point loads the arguments from the stack onto registers and "jumps" to the fast entry point. If it is less then the number of arguments required a closure encapsulating the computation is created.

Eval/apply in GHC is compiled in a similar fashion to push/enter, except that the caller decides how to call the function. This includes how many arguments to apply it to. Again GHC compiles a function with two entry points; a fast entry point for known calls and a slow entry point for unknown calls. A function of arity $n$ can be called with any number of arguments, less than $n$, and thus theoretically we need a separate entry point for every value between 1 and $n$. This could be quite costly, but GHC pre computes a set of entry functions, one for each $N$ that we can enter when we apply the function. Although theoretically this would reduce the number of entry points to $N$, GHC needs different entry points for unboxed and boxed values because the calling convention for the function that the continuation will call may depend on the types of its arguments. Therefore it needs $3^N$ call continuation entry points, one for each data type, if we restrict argument types to pointer, 32-bit non-pointer and a 64-bit non-pointer. An unknown function call consists of query to the callee for its arity and a argument satisfaction check. The appropriate precomputed slow entry point is selected based on this information.

In case of the intermediate language Fleet, described in (Faxn 1996*a*), a number of *partial application descriptors* are generated. A partial application descriptor has the form `papp` $g_1 \ldots g_n$, where $g_1$ are names of global functions, each handling a particular *call signature*. The function `sig` maps a sequence of variables to an integer uniquely identifying the corresponding signature.

In the completely general case for every $k$-ary global function $f$. Fleet

generates a partial application descriptor for `f`, applied to 0 to $k-1$ arguments. Each of these descriptors is an n-element vector of code addresses, each pointing at a *wrapper* function corresponding to a distinct call signature which may occur during the execution of the program. Fleet takes some pains to reduce the cubic code bloat by noting that not all the partial application descriptors are needed.

There are already a number of functional languages that compile to the CLR, including $F^\sharp$ and SML. $F^\sharp$ is a research language produced by Microsoft Research. Since $F^\sharp$ also allows currying and targets the CLR, we were interested in how this is accomplished. $F^\sharp$ targets a set of extensions for the IL called ILX. ILX is similar to the generic extensions already incorporated into .NET. In fact it uses a similar syntax for embedding type application at the level of the IL instruction.

ILX provides a closure class that is used to implement a value of function type. It provides the definition of a single `apply` method. A closure need not correspond to a normal class as the semantics of closures are weaker. For example, the reflection semantics of closure classes are undefined. It also does not contain additional methods, fields, attributes, data, security declarations or any other baggage found in a normal class. Closures that can be partially applied are declared in "curried" form.

Function application is performed using the `callfunc` instruction. This applies one or more groups of arguments. The function value appears first on the stack, followed by the argument groups in sequence. Where the number of arguments is less then the function's arity, determined by a runtime argument check, ILX builds a intermediatory representation to store the supplied arguments and suspend the computation until the remaining arguments are supplied.

ILX provides the instruction `callclo` which is used to make a "direct" call to a closure, similar to a fast entry point in a GHC FUN object. This corresponds to a known function call. Closure declarations introduce corresponding closure types. Type annotations can then be used to show that function values are known to belong to particular closure types. Closure types are primarily offered to optimising compilers; to record evidence that demonstrates that direct calling to known closures is sound, (i.e. known).

The ILX also provides a number of extensions for targeting the Generic CLR. These are discussed in (Syme 2001). They resemble the intermediate type language of GHC based on the Gerard-Reynolds second order calculus, in the sense that it uses explicit type applications to polymorphic functions.

*Compilation without Partial Applications*

Compilation, without runtime argument checking, is simplified in the following ways:

1. all applications are saturated, so we do not require a stack mark to prevent suspensions from consuming to many arguments.

2. by reducing run-time bloat. Because functions are smaller, we no longer have two entry points.

3. by removing any operational distinction between a slow and fast entry point, arguments to previously unknown functions can now be passed in registers.

4. by removing the need for direct stack manipulation we can now target, more mainstream intermediate languages. For example GHC could now target the C– intermediate language.

There are a number of other simplifications which result when removing the requirement for runtime argument checking. The reader can consult (Marlow & Jones 2004) for more information.

With this investigation we are now in a position to frame a research objective. Can we compile a lazy and/or eager functional language to the CLI system stack using the eval/apply evaluation method?.

### *1.3.3 Type Inference*

Milners $\mathcal{W}$ algorithm used in ML (Damas & Milner 1982) constitutes the typical type checking algorithm used by many language implementations. It infers "principle" types in the sense that every other type that can be

derived is a simple substitution of the principle typing. Principle typings are useful because program fragments can be typed independently of their context. Programmers may rest assured that any absent information will be filled in with types that are at least as general, and at least as succinct, as any information the programmer would have provided.

For higher-order languages, the necessary types can become quite complex, and requiring all of the types to be supplied in advance is burdensome. With type inference the compiler takes an untyped or partially typed term and either completes the typing of the term or reports an error if the term is untypable. Any type inference algorithm must be proved decidable for some arbitrary program. This means that it must under all cases be able to decide the appropriate constraint for some term. However, there is usually a trade off of expressiveness verses undecidability of the inference problem.

ML's polymorphism support allows restricted uses of $\forall$ quantifiers. In practice MLs limitations on polymorphic types make some kinds of code reuse more difficult. Programmers are sometimes forced into contortions to provide code for which the compiler can find typings. There are a number of extensions to the basic ML-polymorphism presented by Milner that increase the expressiveness of the type system. This has motivated a long search for more flexible type systems, with good type inference algorithms. Several famous type systems including System F (Girard et al. 1989) and many of its derivatives, have been proved undecidable under type inference (Urzyczyn 1992), (Wells 1993) they therefore do not satisfy the decidability criteria for an appropriate static type checker. Several extensions, including intersection types and second order bounded quantification, have proved to be undecidable (Pierce 1994). Decidable decision procedures, usually involving a finite-rank restriction, however, do exist.

A popular extension is the use of subtyping with polymorphic type inference. Subtyping intuitively allows a value of one type to be used where another type was expected. This is usually expressed using an inequality between two types, called a constraint. Subtyping is a well-researched field, discussed in a number of places (Mitchell 1984),(Fuh & Mishra 1988),(Fuh & Mishra 1989),(Odesky et al. 1999),(Mitchell 1991),(Wand & O'Keefe 1989),(Aiken & Murphy 1991),(Aiken & Wimmers 1992) and (Aiken & Wimmers 1993).

This line of research has mostly focused on complete inference algorithms, which produce principle types. Although complete algorithms for polymorphic subtype inference exist, practical language implementations that take advantage of these are in short supply. The main reason seems to be that the algorithms are inefficient and the output, even for relatively simple input expressions, appears excessively long and cumbersome to read (Hoang & Mitchell 1995).

A constrained type consist of both a "standard type", in the Hindley Milner sense and a constraint set. A trivial example is

$\lambda x.\texttt{x}\ \texttt{x:(}\alpha \to \beta\texttt{)}\ \texttt{|}\ \texttt{\{}\alpha \leq \alpha \to \beta\texttt{\}}$

where $\alpha$ and $\beta$ are type variables and $\alpha \leq \alpha \to \beta$ the constraint set $C$. If we combine this type with universal quantification, we can interpret the lambda as the product of every instance of the type $\alpha \to \beta$; where the constraint $\alpha \leq \alpha \to \beta$ holds. Unlike ML based type systems, where constraints are equalities, "solving" a set of constraints becomes a problem in deciding if the "closure" of the set exists. Pottier (Pottier 2001) and (Trifonov & Smith 1996) extends this by investigating subtyping between constrained types. Given the constraint set $C$ and the type $\tau$, which is satisfiable within $C$, can we use a type $\tau \mid C$ where the constraint set $C'$ and type $\tau'$ are expected? This is called *entailment*, which is currently undecidable. Inference algorithms for subtyped systems produce constraint sets that are large and give little comprehensible feedback to the user. Pottier (Pottier 2001), (Pottier 1998) gives a number of transformations that seek to make subtyping practical, by providing ways to reduce the complexity of the subtype constraints; which can be exponentially large for even a simple program. Still, writing a usable compiler that incorporates these transformations is a *large* undertaking, requiring a number of additional steps. These include constraint set simplification, (some tailored for different subtype models), like garbage collection, minimization and canonization, discussed at some length in (Pottier 2000) and (Trifonov & Smith 1996). It is also difficult to tailor constraint based subtype algorithms for inferring types across intermodule dependencies, which would hamper the practical uses of the Mondrian compiler. Many attempts have been made at simplifying the output from these algorithms

but they have only partially succeeded. The problem in its generality seems to be intractable, both in theory and practice (Trifonov & Smith 1996).

However, even if type simplification were not an issue, there is an inherent conflict between generality and succinctness in polymorphic subtyping, that is not present in the original ML type system. While the principal type of an ML expression is also, syntactically, the shortest type, the existence of subtype constraints in polymorphic subtyping generally makes a principal type longer than its instances. In particular, the principal type for a given expression may be substantially more complex than the simplest type general enough to cover an intended set of instances. Type annotations, which give the programmer direct control over the types of expressions, are therefore likely to play a more active role in languages with polymorphic subtyping than they do in ML, irrespective of advances in simplification technology.

The subtype relation in the $\mathrm{CLR}_{\leq}$ and $\mathrm{CLR}_{\leq,\forall}$ is restrictive in comparison to the more powerful forms of subtyping we have discussed. Unlike the usual subtype relation which is based on structural inequivalence, it is based on name inequivalence. For instance, the subtype relation defined between arbitrary record types by Remy (Remy n.d.) uses a form of structural decomposition. Name equivalence allows the subtype relation to be defined by incremental construction of polymorphic records and data types, using inheritance. This is more suited to languages like $C^\sharp$ and C++.

In light of the complexities of a subtyping, we restrict the subtype problem to inferring implicit coercions from subtypes to supertypes. This means, we can keep most of the usual ML unification algorithm from previous work, and produce a useable compiler solving the problem of partial applications and type inference with subtyping, within timeframe suitable for a masters thesis. This approach is also used by Nordlander in (Nordlander 1998), who presents an inference algorithm that always infers types without subtype constraints (if it succeeds). Nordlander uses a method based on unifying constraints, generated, where appropriate, to reduce the complexity of the inference problem. This algorithm favours readability over generality, leaving it to the programmer to provides type annotations where this strategy is not appropriate. We adopt a similar strategy, though our solution is generally less powerful.

In the context of this research we ignore recent advances in the CLI. Microsoft have released a new version of the CLI that has extends the power of the subtype relation over the $\rightarrow$ operator, which we call $\text{CLR}_{\leqslant,\forall}$. $\text{CLR}_{\leqslant,\forall}$ observes the usual "contra/covariance" over function types which is lacking in the previous versions. We can consider $\text{CLR}_{\leq}$ and $\text{CLR}_{\leq,\forall}$ as subsets of $\text{CLR}_{\leqslant,\forall}$ and restrict our discussion to these runtime targets.

We are interested in building a practical type inference algorithm capable of type checking the Mondrian language, which constitutes our second research objective. Mondrian imposes a few difficulties when using the Hindley Milner algorithm. Unlike ML where data constructors are disjoint, constructors are related by a subtype relation in Mondrian. This is explained in more detail in the following chapter. This complicates inference over conditional expressions, because when using reference types the type of the conditional must be the least upper bound of the types of the case expressions.

## *1.4 Contributions*

This thesis makes the following contributions.

### *1.4.1 Removal of Runtime Argument Checking*

We discuss the feasibility of a transformation that allows us to remove runtime argument checks by transforming a curried language into an uncurried one, with explicit closure creation.

We characterise a set of annotations made to inferred types to give us a faithful model of reduction. These are termed operational types or *o*-types and are a novel way of annotating reduction information within the type system.

With a faithful model of reduction we are in a position to describe a set of transformations which saturate all function calls. This effectively removes all partial applications and allows us to develop a machine targeting the CLI system stack. This transformation we call *lambda doping*. It lifts known and unknown partial applications. This transformation was inspired by (Perry 2002*a*).

### *1.4.2 Type Inference on the CLI*

We describe a type inference process that infers principle types for the language Mondrian. It uses a name inequivalent subtype ordering that infers implicit coercions where appropriate. The inference algorithm prefers strictness of generality in the sense that it never infers types with subtype constraints. Unlike the more expressive systems described earlier, this system constrains the use of subtyping by only inferring widening coercions.

We augment the basic inference algorithm by generating operational types. This strategy is used by the lambda doping algorithm.

### *1.4.3 The Compiler*

We describe the implementation of a compiler using a eval/apply reduction model. The implementation of this machine is simplified by a lack of requirement for runtime argument checking and the generation of *o*-types. *o*-types give us a easy way of building the reduction pathway.

# Chapter II

# The Mondrian Language

A Mondrian program consists of a set of combinator definitions with a distinguished definition `main` specifying the start point. Mondrian lacks "impure" constructs: both linguistics (e.g. updatable state, exceptions, or first-class continuations) and external interaction (I/O, OS interface, etc.) are implemented via a monadic discipline (Peyton Jones & Launchbury 1995).

Mondrian supports the usual ML language semantics for declaring local expressions and anonymous lambdas, but deviates in its implementation of data constructors. It provides two forms of let bindings for non-recursive and recursive definitions named `simplelet` and `let`, respectively. We limit this discussion of Mondrian and we focus on a subset of the language, namely the usual "functional" aspects and interaction with .NET.

## *2.1 Language considerations for .NET*

Mondrian is designed as scripting language for .NET, and thus, its design is biased towards easy binding with other .NET languages and libraries. Mondrian takes the following approach:

- It provides the ability to consume .NET defined structures, for example we can invoke methods on classes in disjoint namespaces

- It takes a minimalist approach to providing the ability to define .NET equivalent structures within the language. For example, it does not provide the ability to create nested method definitions within class structures.

| .NET type | $C^\sharp$ type | Mondrian type |
|---|---|---|
| System.Boolean | bool | Boolean |
| System.Char | char | Char |
| System.Double | double | Double |
| System.Single | float | Real |
| System.Int32 | int | Integer |
| System.Int64 | long | Long |
| System.String | string | String |
| System.Object | object | Object |

Table 2.1: Correspondence between types in .NET and Mondrian

### *2.1.1 Namespaces*

A namespace is designed to segregate functions into groups of related functionality. Mondrian supports the equivalent construct through the `package` statement. This is translated directly to its namespace equivalent.

### *2.1.2 Values*

Values types in .NET are sealed and cannot be subclassed. They do not need to be boxed on the heap. Value types, like ordinary classes, can have fields and instance methods and can be viewed as structured primitive types. However, because value types are sealed, they do not need to carry runtime type descriptors, which are used to support checked downcasts and virtual method invocations. Every value type derives from the base class `System.ValueType`, and observes copying semantics, not by reference. Every primitive type in Mondrian has a corresponding type given by its CLI equivalent. All of the simple types are aliases of the .NET Framework System types. For example, `Integer` is an alias of `System.Int32`. A complete list is given in Table 2.1.

### *2.1.3 Algebraic Data Constructors*

Mondrian like most other modern languages, supports *structured data types*, or algebraic data types *encoded* as their isomorphic equivalents on the CLI. Mondrian, like ML, structures recursion explicitly over data types. Meijer

et. al. (Meijer & Claessen 1997) describes this transformation in terms of a isomorphism between Haskell data types and their equivalents in Mondrian. Stated informally, a Haskell data type declared using the `data` keyword can be encoded in Mondrian by observing the natural bijection between data type labels and class names. For every data type label `D`, we can create a class `D'`, then for every type constructor `C` declared in `D`, we can construct a class `C'` which is *derived* from the class `D`. The fields of the type constructors are naturally encoded in the class definitions.

The Haskell definition defined in Listing 2.1

```
data Maybe v = Nothing | Just v

data FMBT k v
    = E
    | BT (FMBT k v) k (Maybe v) (FMBT k v)
        deriving (Show, Read, Eq)
```

Listing 2.1: `BinaryTree`, defined in Haskell

demonstrates the use of a recursive domain definition. We can readily describe recursive definitions in both Haskell and Mondrian. The Haskell definition can be rewritten in Mondrian as:

```
class Maybe {};
class Nothing : Maybe {};
class Just : Maybe {  Object value; };

// BinaryTree definition
class FMBT  {};
class E : FMBT {};
class BT : FMBT {    FMBT left;
                     Object k;
                     Object v;
                     FMBT right;
                };
```

Listing 2.2: `BinaryTree`, defined in Mondrian

While both data structures describe a binary tree, the definition given in Mondrian lacks the type parameters `k` and `v` used in the definition for Haskell. Type parameters allow the Haskell `FMBT` definition to be specialized,

for example, to `FMBT<Int, Int>` which will create a `FMBT`, which stores keys of type `Int` and values of type `Int`. The same affect can be achieved in Mondrian by using a value of type `Object` for the values `k` and `v`. This allows `FMBT`, defined in Mondrian, to store values of any type. This is an example of subtype polymorphism, as opossed to parametric polymorphism, used in Haskell. It works because the $\text{CLR}_{\leq}$ defines a supertype `Object`. However, while we gain in expressitivity we loose in performance, everytime a value of type `Object` is used a runtime check must be performed.

The reason for this is that Mondrian compiles to the $\text{CLR}_{\leq}$ which does not provide any of encoding parameterised data structures. We will look at extensions to the Mondrian language which support parameterised data types in section 7.2.2. The list data type in Haskell can be defined as:

```
data List a = Nil | Cons a (List a)
```

Listing 2.3: List, defined in Haskell

In Mondrian this is given as:

```
class List {};
class Nill : List {};
class Cons : List { Object head; List tail};
```

Listing 2.4: List, defined in Mondrian

Here we manipulate the `BinaryTree` defined in Listing 2.2. This code snippet assumes that tuples are represented using the `Pair` data type, which is defined in `mondrian.prelude` and reproduced here from section 2.5 for clarity, with the appropriate projection operations, `fst` and `snd`. We can use `switch` expressions to pull apart data structures. Note that even though Mondrian does not include pattern matching, it has special syntax handling to translate `x::xs` to the equivalent `Cons { x = head; xs = tail;}`.

```
class Pair
{   Object : t1;
    Object : t2;
};

fst = p ->
    switch(p)
    {   case Pair{x = a; y = b;} : x;
```

```
    };

snd = p ->
    switch(p)
    {   case Pair{x = a; y = b;} : y;
    };
};
```

Listing 2.5: Pair, fst and snd, defined in Mondrian

If we were working with parameterised data constructors, then the function `makeRandomPairIntList` would return a list of values of type $Pair < Integer, Integer >$, representing `(Key, Value)` tuple pairs. However, at present, it will return a list of Objects typed as $Pair$ which hold object references to values with runtime types of $Integer$.

The data is added to the `BinaryTree` by using the `insertFMBT` function. To this function, we supply the key and value projected from the `Pair` by using the `fst` and `snd` projection functions. Finally, we print the `BinaryTree` by iterating over the nodes in a depth first manner using the function `mapAccumF`. This function is defined in the usual way, however, we accumulate over a tree instead of a list. `mapF` is also defined in a like manner, that is, we map over a tree. A similar result could be achieved by flattening the `BinaryTree` to a list and iterating it.

We also give the types of the functions `insertFMBT`, `mapF`, and `mapAccumF` below. This is to save space defining the functions explicitly. However, at present, Mondrian does not support user type annotations or generics. Their extension to the language is discussed in section 7.2.1.

Consider the following:

```
//makeRandomPairIntList : [Pair<Integer, Integer>];

//makeRandomList generates a random list of numbers.
buildBinTrie
 = let randPoints = makeRandomList;

       //Add every number to the BinaryTree, accumulate
       //the intermediate value in Pair<>
       btrie
        = fst $ mapAccumL
```

```
                (acc -> v ->
                  let k = fst v;
                      v = snd v;
                  in  new Pair { a = insertFMBT k v acc; b = v }
                ) emptyTrie randPoints;

    //Generate a string of (Key, Value) pairs by traversing the
    //nodes and using the string concatenation operator `++`
    str
     = fst $ mapAccumF (acc -> v ->
                          new Pair { a = acc ++ (show v); b = v)})
                             [] btrie;
in  {    // Map through the items of the BinaryTree using the
         // BinaryTree MapF function
         putStr ``BinaryTree items\n``;
         putStr str;
    };
```

Listing 2.6: Using data structures

Mondrian supports the usual semantics of data projection by using the `switch` statement to break up data structures. `Switch` uses the runtime type of the value to decide on which case alternative to execute. We show the use of `switch` in 2.5, here `switch` projects a `Pair` structure. This example is rather trivial as there is only one data constructor for the pair data type, `Pair`. In the `case` alternatives, we use bindings of the form `identifier = <expression>`. A field binding is very much like a normal binding found in a `let-` or `where`-clause. In a pattern match, the equation `n = name` binds the variable `n` to the actual value of the field name. In a construction or update, the equation `name = "John Doe"` binds the field `name` to the string `"John Doe"`. Field bindings are recursive, thus for example `Student {name = name}` binds name to $\perp$, as it assigns the value of the name field of instance under construction to the name of the field of the instance under construction.

*Exploiting the Subtyping Relation*

Unlike ML, Mondrian allows the definition of data types with overlapping sets of constructors. Earlier we remarked that there is a natural correspondence between Haskell data types and those defined in Mondrian. In truth,

subtyping allows us a little more freedom in defining the data type. Consider this example:

```
class Worker { name : String;
               age : Integer;
               address : String;
             };
class Waiter  : Worker { bar : Boolean;} ;
class Trucker : Worker { loggedhrs : Integer;
                         dangerousGoods : Boolean;
                       };
```

Listing 2.7: Worker hierarchy

While the example in Listing 2.7 is rather contrived, it does demonstrate some of the benefits of exploiting the subtype relation for describing the sum type. The fields `Name`, `Age` and `Address` are common data elements shared over all classes derived from `Worker`. This blurs the definition of a sum type, as the elements are tagged with the `Worker` class type. We can also promote a subclass of `Worker` to a `Worker` class and therefore access the class internals as if the type of the value was just a `Worker`.

The following code exploits this by retrieving the fields `name`, `age` and `address` from the supplied worker, which may be a `Waiter` or another `Trucker`, and supplies them to the new `Trucker` object.

```
assignTrucker : Worker -> Integer -> Boolean -> Trucker;
assignTrucker wrkr hrs df
    = switch wrkr of
      {     case Worker of
            {   name2 = name;
                age2  = age;
                address2 = address;
            }
      :     {   new Trucker { name = name2; age = age2;
                              address = address2;
                              loggedhrs = hrs;
                              dangerousGoods = df;
                            };
            };
```

```
};
```

Listing 2.8: `assignTrucker` function definition

### *2.1.4 Product Types*

Mondrian at present does not support "pattern matching" on data values in the traditional ML sense. This limitation makes the use of product types a little difficult. As product fields are not labelled we cannot deconstruct them using the case notation discussed above. Given the expression `let a = (1, 2) in ...`, Mondrian will construct a value of type `Pair`, which is defined as:

```
class Pair {Object a, Object b};
```

The use of `Pair` was introduced in Listing 2.6, and its manipulation here mirrors that of Lisp, where the functions `fst` and `snd` return the first and second elements, respectively. The second element can of course be another `Pair` value, which leads to constructions such as `let a = (1, 2, "hello", [2,3])`, and accessing the 3rd element becomes `fst(snd(snd a))`.

## *2.2 Interacting with .NET*

Mondrian binds to functions declared by other languages within the IO Monad idiom. We use the IO Monad to effect side-effect free interaction with the world, in the same manner as GHC (Peyton Jones & Launchbury 1995). The IO Monad is a state transformer, with the state being the `world`. Mondrian provides a number of functions that interact with the world over the IO Monad, including `invoke` and `invokeStatic`, which are discussed in more detail below.

### *2.2.1 Creating Objects*

We can create object instances of type $\tau$ using the `create` expression. The example below constructs a object of type `Random`. The qualification is necessary if we do not explicitly import the namespace into the local binding context. In the following, we create an instance of a `Vector` and parameterise it to type `Integer`, using the following expression:

```
vector <- create Vector(Integer) NUMBER;
```

### *2.2.2 Reference Semantics*

Mondrian provides a kind of SML like reference semantics without using impure extensions. We can mimic the following SML fragment in Mondrian by using specialized functions that manipulate variables of type `Var<X>`, where `X` is some type variable.

```
g = let val r = ref (Pair(1,2)) in r.#invert();!r end,
```

In the example below we manipulate the variable x defined as `Integer` within a `Var` object

```
f : Var<Integer> -> IO<Void>;
f = x -> { setVar x 2;};

g : IO<Void>;
g = { v <- create Var 0;
          setVar v 2;
          putStr (show $ getVar v);
          f v;
          putStr (show $ getVar c);
    };
```

Listing 2.9: Manipulating state

`g` creates a new `Var` and initialises it to 0.

### *2.2.3 Fields, Methods and Properties*

We can bind static, per-class, fields and methods to function bindings by using `invoke` or `invokestatic` expressions. `Invoke` is syntactic sugar used by Mondrian to use the reflection libraries to query for method data on supplied class instances. `Invoke`, like any function application, can be curried and therefore we can implement an interesting form of dynamic binding. The `invoke` call specifies the function to call and its type signature, but we can delay supplying the parameters and the object instance. The following example shows how we can use invoke to bind the method `Split` taken from the class `System.String`. `System.String.Split` is defined in $C^\sharp$ as

```
public static string[] Split(char[] separator, int count);
```

which can be interpreted as the type

```
Split :: System.Array Split (System.Array, Integer);
```

and bound appropriately as

```
str2 <- invoke System.Array.Split (System.Array, Integer) sep 4 str
```

where `str2` is the `String` instance. We assume that `sep` has been created using the appropriate array creation functions. The `invokeStatic` method is used to invoke static methods that do not require object instances. Its binding semantics differ from `invoke` only in the number of arguments it takes. The `invoke` method takes as many arguments as there are types provided in the definition. In the example above, it is typed with two function types `(System.Array)` and `Integer` and is therefore provided two arguments and an object instance.

Mondrian provides special methods for directly binding and manipulating properties or fields. These are called `get` and `set`.

# Chapter III

# Compiler Overview

The Mondrian compiler compiles the Mondrian term language, a subset of which we discussed in the previous chapter, to IL. We distinguish between the different versions of the Mondrian compiler in the following way. The original Mondrian compiler is called Mondrian I. It compiles to the Unsaturated Mondrian Exceptional Machine ($u$-MEM), running on the $\text{CLR}_{\leq}$ platform. Mondrian I is described in (Perry 2002*b*), (Perry & Meijer 2004), (Meijer et al. 2001) and (Smith et al. 2002). It has one machine target and one platform target, the $u$-MEM and the $\text{CLR}_{\leq}$, respectively. The original $u$-MEM handled unsaturated applications by performing runtime argument checks. Evaluation is described on the $u$-MEM in some detail in section 4.3 for comparison.

The new Mondrian compiler, Mondrian II, is distinguished by its use of type inference to assign types to terms, and its use of this information to effect type transformations that saturate all function calls. We extend the transformation pipeline by incorporating a new transformation, called lambda doping, that saturates all function calls, in both *known* and *unknown* situations. We also extend the machine and platform targets. Mondrian II can target the Saturated Mondrian System Machine ($s$-MSM). The $s$-MSM is an eager machine, for reasons discussed in section 1.2.1, and runs on the platforms $\text{CLR}_{\leq}$ and $\text{CLR}_{\leq,\forall}$.

## *3.1 The Mondrian Transforming Compiler*

The Mondrian compiler is logically divided into three parts. Each of the parts consists of a number of passes (we use the word *pass* to mean a single traversal of the abstract syntax tree):

- The **front-end**, which parses, preinfers, type checks/infers and desugars the program;

- The **middle**, which performs lambda doping and lambda lifting;

- The **back-end**, which translates the typed Mondrian program into abstract machine code, in this case we use $C^\sharp$.

With saturated function application, we now have the ability to target portable high-level intermediate languages, such as $C^\sharp$ and Java. By target, we mean using the language's stack, instead of creating a new runtime abstraction in the language, which is currently what GHC does when it targets C. Portability for any language is restricted by the existence of a compiler, or, in this case, a runtime environment for the CLI. This is not a problem because there are currently implementations of the CLI running on Linux, FreeBSD, and Windows.

### *3.1.1 The Front-End*

Mondrian II parses the Mondrian language from which we define a subset called $\mathrm{M}_{CM}$, given in Definition 3.1 producing a parse tree. We do not use a separate intermediate language, but apply transformations over the Mondrian term language. We use the parse tree to store any additional annotations that are added to the term language. In our case, the annotations are added in two locations. To each token, parsing adds the physical location in the source file and type inference adds the inferred type. Recall that Mondrian I is dynamically typed and therefore does not annotate terms. We refer to the type unannotated term language as the $M$ calculus. Type annotated Mondrian terms are referred to as the $M_{\leq}$ calculus. Where the use of different calculii is obvious by context we drop the subscript. This is an idea similar to that employed by the GHC compiler, which uses a intermediate representation language, that it calls the *Core* language, discussed in more detail in (Jones 1996) and (Peyton Jones 1992). The Core language is the based on the Girary-Reynolds 2nd order lambda calculus System F, (Girard et al. 1989), as such it stores type information explicitly. For example, the

function `mapAccumL`, is a polymorphic function with 2 universal quantifiers $A$ and $B$, giving us `mapAccumL` defined as $\Lambda A.\Lambda B.\lambda f.\lambda xs.e$. This is then applied to the appropriate types inferred by type inference.

We also perform a number of preliminary passes which we term "preinference". These passes include the following: a check that all types are visible and storage of top-level type declarations in the symbol table.

Mondrian supports module level compilation by generating an interface file denoted by a `.mi` for each `package` compiled. An interface file contains information such as the inferred type of each exported function plus versioning information. When compiling any `package` that imports `P`, Mondrian consults `P.mi` to extract information about exported functions from package `P`.

*The Core Mondrian Language*

The Mondrian Core term language ($\mathrm{M}_{CM}$) is defined in Definition 3.1. It is a minimal subset of the Mondrian language. We use $\mathrm{M}_{CM}$ for most of our presentation as it simplifies the discussion.

**Definition 3.1.** *We use the syntax $\overline{x_i}$ to mean some non-empty list of $x$ indexed by $i$. Where we use the superscript $\overline{x_i}^*$, we mean that the list can be empty. We use the type variable $\alpha_i$ to type the class field $v_i$. We use $T$ for the data type declared by the class declaration, and $T_1$ for the type from which the class is derived.*

$$
\begin{array}{rrcl}
\text{(declarations)} & d & ::= & \text{class } T\ T_1\ \overline{v_i : \alpha_i} \mid \text{import} \mid v\ =\ e \\
\text{(expressions)} & e & ::= & v \mid e_1\ e_2 \mid \lambda \overline{x}.e_1 \mid \\
 & & & \text{let } \overline{x_i\ =\ e_i}\ in\ e_2 \mid \\
 & & & \text{simplelet } \overline{x_i\ =\ e_i}\ in\ e_2 \mid \\
 & & & \text{switch } e\ \overline{\text{case } v_i\ \overline{(x_{ij}\ :=\ e_{ij})^*} => e_i}\ \text{default } e_{def} \mid \\
 & & & \text{new } v\ e \mid \\
 & & & \text{if } b \text{ then } e_1\ e_2 \mid \\
 & & & \text{Closure } c\ e_1\ v\ \overline{e_i}\ v\ v \\
\text{(values)} & v & ::= & str \mid c \mid x \mid
\end{array}
$$

The set of free variables in an expression $e$ is given as $FV(e) \cap \mathcal{V} \neq \varnothing$ and defined in the usual way. It is a subset of the set of all variables $\mathcal{V}$. For the most part, the language is consistent with the literature. Class declarations construct sum types, which are sets of values and their associated types. We can extend a class from a super type by using the syntax:

```
class A : B {};
```

`A` and `B` correspond to $T$ and $T_1$ respectively, in the grammar. We can construct values of sum type, given in Listing 2.8, using the `new` expression. Finally, we extend the language with a Closure, which we use to encapsulate expressions.

### *3.1.2 The Middle*

The middle is responsible for a number of optimisation and translation steps. These are independent of both the source language, as they are applicable to a wide variety of functional languages, as well as the target machines. There are a number of transformations that fit into this phase, including strictness analysis and deforestation (Gill et al. 1993).

The Mondrian II compiler currently performs two transformations: the routine lambda lifting transformation, which we discuss below, and the lambda doping transformation, which we discuss in chapter 6. Lambda doping is used to saturate all function application. This allows us to create runtime implementations that do not require runtime argument checking. We discuss the interaction of lambda doping on two common transformations, namely, full laziness and strictness analysis in section 6.4.

#### *Lambda Lift*

The Lambda lift is used to capture free environment in a Closure. It is accomplished in the usual way. We use a Closure to store the free environment. For example, the following

```
f = a -> let f1 = a -> f2 a;
             f2 = a -> a + 1;
         in  f1 a;
```

is converted to

```
f = a -> let f1 = Closure "f1" (a -> f2 a) {f1, f2} 1 1;
             f2 = a -> a + 1;
         in  f1 a;
```

The Closure stores the free environment `f1` and `f2`. The pending and argument counts of `1` are included because we use a Closure to encapsulate partial applications as well, but there is no operational distinction between the two. Currently, the algorithm does perform ecursive binding analysis. If it did then we can further transform the expression to

```
f = a -> let f2 = a -> a + 1;
         in  let f1 = Closure "f1" (a -> f2 a) {f2} 1 1;
             in  f1 a;
```

Note that in this example the Closure is called "f1". Where the Closure is not bound to a let binding, we will often omit the string identifier.

### *3.1.3 The Tail-End*

The Tail-End is responsible for translating from the Mondrian term language to the intermediate language GooG, and from GooG to $C\sharp$. GooG is an object-oriented high-level grammar which provides a convenient way of pretty printing to $C\sharp$ or Java. The Tail-End also deals with translating the type language, inferred in the Front-End phase to the different runtime targets, discussed in section 7.5.

It would be possible to skip GooG and pretty print from the Mondrian term language to $C\sharp$, but compiling via GooG allows us to easily extend the pretty printing output targets. For example, we could easily write a pretty printing algorithm to VB.NET from GooG. We do not discuss GooG any further for simplicity but note its existence here.

## *3.2 Runtime Library*

To complete the overview, we give the class library in Figure 3.1. This is just a small selection to gain an appreciation of the library internals. Some of these functions are detailed further when we discuss the compilation to the *s*-MSM.

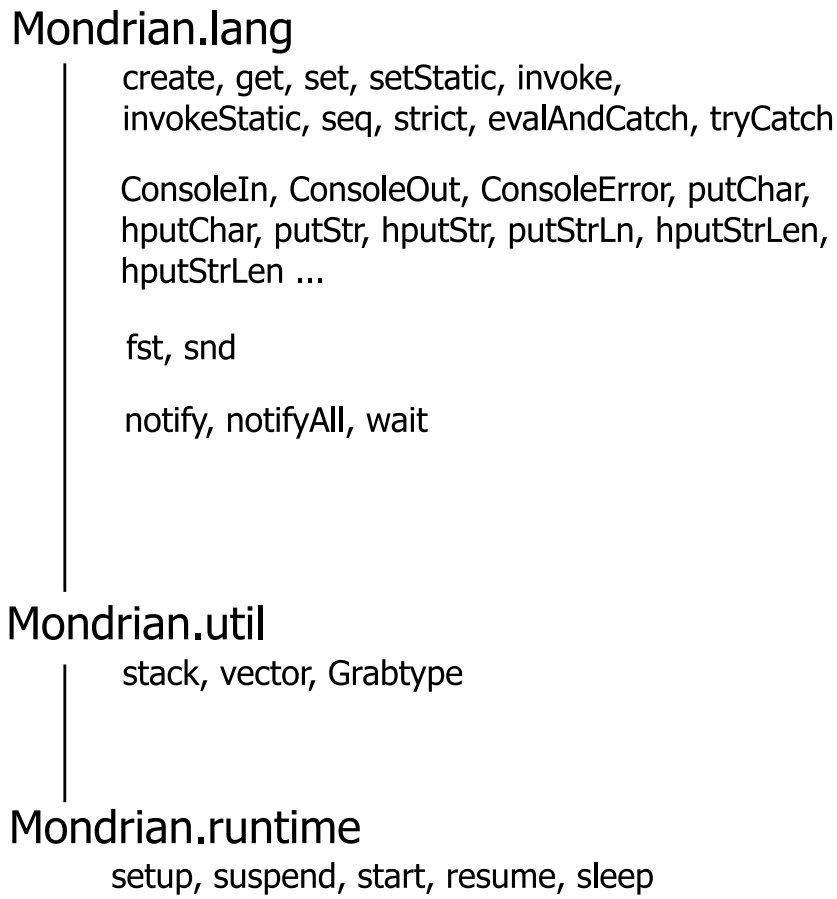


Figure 3.1: Runtime library

### *3.3 Presentation of Discussion*

Our discussion of the three sections of the Mondrian II compiler are not in operational order. This is required to discuss the need for specific type information, which allows us to effect the lambda doping transformation, before we can discuss how we can infer it.

Our discussion will take the following form:

- Operational Semantics for the $u$-MEM and $s$-MEM
- Reduction without Runtime Argument Checking
- Middle : The Lambda Doping Transformation
- Front-End : The Type System
- Tail-End : Compiling to the $s$-MSM

# Chapter IV

# Compilation Model

We are interested in developing a set of syntax directed reduction semantics for eval/apply evaluation on the system stack that do not rely on runtime argument checking. The MEM implements reduction via push/enter and the MSM to implement eval/apply reduction. Implementation details for both are described in section 4.3 and chapter 8 respectively.

We will give the formal operational semantics for $u$-MEM and $s$-MSM. To save notational burden, we will use MSM and MEM in this discussion to refer to the Saturated Mondrian System Machine and Unsaturated Mondrian Virtual Machine, respectively. We will look at the feasibility of a saturation transformation in the following chapter.

Having defined the reduction semantics, we discuss how the heap objects are compiled to $C^\sharp$. We will then give an overview of evaluation in the $u$-MEM.

Early evaluation mechanisms were based on maintaining the natural structure of the parsed program which was usually a tree. While theoretically elegant, tree traversal mechanisms are slow and bulky. Reduction requires the copying of contractums in place of redexes into the expansion tree and there is too much backtracking during tree traversal. Mondrian, as does GHC, uses a stack based machine as an evaluator. Both are based loosely on the spineless, tagless g-machine (STG). In the mvm an expression is translated into a sequence of machine instructions [1] that describe a leftmost-outermost reduction (lazy evaluation) of the expression. In the MSM reduction is strict which means redexes are reduced before application. The traversal and reduction steps can be translated into machine code because function expressions are

[1] By way of $C^\sharp$ in Mondrian

statically scoped so environment maintenance is routine, and because the traversal path through the expression can be calculated from the structure of the expression. Thus, substitutions over terms describes a complete rewrite system. We do not discuss the difference between lazy and strict evaluation in any detail, the rewrite mechanism to saturated application is reduction agnostic.

### *4.1 Operational Semantics of MEM and MSM*

The operational semantics for the $\mathrm{M}_{CM}$ language are given in small step form. We "solve" the name-capture problem for reduction by substitution by using closures, each denoted as CLO, to store free environment. Function values, if they contain free environment, are compiled to CLOs, while those that do not are denoted by FUN. Sharing is maintained easily enough by using a heap to store mappings of addresses to CLO/FUN objects. This is sometimes referred to as the *environment* based approach, and is semantically identical to using a *graph* to store a mapping of addresses to Closures, which is used in Faxn (Faxn 1996*a*).

The *machine* is made up of a tuple $M = \langle E, S, H \rangle$, where $E$ is the current expression being evaluated, and $S$ is a stack of pending arguments and call continuations. We use the term "call continuation" in $\beta$-reduction to describe a list of pending arguments representing the "next" thing to be evaluated. For example, given the expression `f 2 3`, where `f` is a single argument function, the call continuation after consuming the first argument is `3`. Call continuations are described using $(\bullet\ a_1 \ldots a_n)$. $H$ is the heap and it can be regarded as a finite mapping from variables to heap objects of the form $x_1 \mapsto OBJ_1 \ldots x_n \mapsto OBJ_n$, where OBJ is some heap object. Heap objects are allocated via `let` expressions, where the right hand side of a `let` is a *heap* object.

In general, there are four kinds of heap objects used by the MSM and MEM:

FUN $(f : x_1 \rightarrow \ldots x_n \rightarrow e)$ is a function closure of arity n, which takes some n number of arguments $x_i$ and a body $e$. All functions are compiled to FUN objects, except where the function contains free environment,

in which case we generate a CLO which carries free environment as a payload. This is discussed in section 4.2.1.

CLO $(C\ x_1 \to \ldots x_n \to e)\ (\dot{}b_1 \ldots b_t)$ creates a Closure. Closures are used to hold *pending* expressions, supplied arguments and a free environment. This is discussed in section 4.2.2. We use $(\dot{}b_1 \ldots b_t)$ to denote the encapsulated arguments.

CON $(C\ a_1 \ldots a_n)$ creates an algebraic data value in which we have the saturated application of arguments $a_i$ to some constructor $C$. This is discussed in section 4.2.3.

THUNK $(e)$ represents a thunk or suspension, caching the value $e$. When the expression $e$ is evaluated to Weak Head Normal Form (WHNF), the resulting value $v$ is used to overwrite $e$. This updates the thunk, so that the next time it is called, the value $v$ is returned. The thunk is self-updating, which means that it handles the update instead of the caller.

A partial application (PAP) is functionally similar to a CLO, in that it also encapsulates expressions and awaits pending arguments. Therefore, we can use a PAP or CLO interchangeably in the discussion. FUN and CON can be considered heap *values* or canonical forms and cannot be evaluated any further.

We can rewrite the `map` function in Mondrian using Heap "annotations", given in Listing 4.1:

```
map = f -> l ->
    FUN switch l
        {   NIL     -> nil;
        ;   CONS { x = head;
                   xs = tail;
                 }  -> let h = THUNK (f x);
                           t = THUNK (map f xs);
                           r = CON (Cons h t);
                       in  r;
```

```
};
```

Listing 4.1: `map` heap annotated

To evaluate this function, we enter the FUN and apply the evaluation rules in Definition 4.1 and Definition 4.3.

We produce equations of the form $e_1; s_1; H_1 \Rightarrow e_2; s_2; H_2$, which describe one step in the evaluation process. An argument for the MEM may be unevaluated. We indicate this by using arguments of the form $Code\ a$. This means we may at some point reduce $a$ to some value.

A Closure stores a partially applied expression with some number of pending arguments. A partially applied expression may consist of some number of totally applied components. A Closure stores some expression $e$ expecting some $n$ number of remaining arguments and some $t$ number of supplied arguments. Supplied arguments are indicated by $b_1 \ldots b_t$, given as CLO $((x_1 \ldots x_n) \rightarrow e)\ (b_1 \ldots b_t)$. Closures also store two numbers representing the number of pending arguments given by $t$ and the total number of arguments required by the expression (this is always $n\ +\ t$).

Following (Marlow & Jones 2004), we use the superscripts $k$ and $\bullet$, applied to the function as $f^n$ and $f^\bullet$, to delineate the arity of a particular function. If the arity is known statically, it is given with a $n$ and with a $\bullet$ otherwise. We first give the reduction semantics common to both the $u$-MEM and $s$-MSM for $\mathrm{M}_{CM}$ in Definition 4.1. In eval/apply, the call continuation is represented using $(\bullet\ a_1 \ldots a_n)$.

The rules are mostly self-explanatory. Rule LET adds a new heap object $obj$ to the heap. It uses a fresh name $x'$ for $x$ in $e$ and the body of the `let`. It then reduces the expression $e$ after substituting $x'$ for instances of $x$ in $e$.

$C^\sharp$ provides a number of higher level constructs such as `switch` and `if` statements that allow us to express these constructs directly and rely on $C^\sharp$ to compile the appropriate IL instructions. However, these are just syntactic sugar and we can compile these directly to equivalent IL instructions.

A *switch* is evaluated when the value of the expression $e$ in $switch\ e\ of$ ... is evaluated and the appropriate switch alternative branch is taken. Switches in Mondrian are a little more complicated because we match on cases differently for value types and reference types. This issue is discussed in more

$$
\begin{array}{rcll}
 & & & \text{(LET)} \\
let\ x\ =\ obj\ in\ e;\ s;\ H & \Rightarrow & e[x'/x];\ s;\ H[x \mapsto obj] & \\
 & & x'\ fresh & \\
 & & & \text{(SWITCH)} \\
switch\ e\ of\ \{\ldots\};\ s;\ H & \Rightarrow & e:\ switch \bullet\ of\ \{\ldots\};\ s;\ H & \\
 & & & \text{(CASECON)} \\
switch\ r\ of\ \{\ldots; case\ Cx_1 \ldots x_n \rightarrow e; \ldots\};\ s;\ H & \Rightarrow & e[a_1/x_1 \ldots a_n/x_n];\ s;\ H & \\
H[r \mapsto CON(Ca_1 \ldots a_n)] & & & \\
 & & & \text{(CASEDEFAULT)} \\
switch\ r\ of\ \{\ldots; default:\ x \rightarrow e\};\ s;\ H & \Rightarrow & e[v/x];\ s;\ H & \\
 & & & \text{(IF)} \\
if\ e\ then\ e_1\ else\ e_2;\ s;\ H & \Rightarrow & e:\ (if \bullet\ then\ e_1\ else\ e_2);\ s;\ H & \\
 & & & \text{(IFTRUE)} \\
True:\ (if \bullet\ then\ e_1\ then\ e_2);\ s;\ H & \Rightarrow & e_1;\ s;\ H & \\
 & & & \text{(IFFALSE)} \\
False:\ (if \bullet\ then\ e_1\ then\ e_2);\ s;\ H & \Rightarrow & e_2;\ s;\ H & \\
 & & & \\
 & & & \text{(PRIMOP)} \\
\oplus\ a_1 \ldots a_n;\ s;\ H & \Rightarrow & a;\ s;\ H & \\
 & & where\ a\ is\ the\ result\ of & \\
 & & applying\ the\ primitive & \\
 & & operation\ \oplus\ to\ arguments & \\
 & & a_1 \ldots a_n &
\end{array}
$$

Figure 4.1: Reduction semantics common to both the $u$-MEM and $s$-MSM

detail when we discuss how the switches are translated to the CLR in section 7.2.2. However, in both cases, a switch reduces its scrutinee to a value and we switch on the value, be it of reference or value type.

**Definition 4.1.** *We use the $M_{CM}$ grammar to define the reduction semantics in Figure 4.1 by case analysis over each term.*

**Definition 4.2.** *u-MEM $\beta$ reduction rules are defined in Figure 4.2(a).*

- PUSH is used to push applied arguments onto the argument stack.
- FEXACT is used when the function f is a FUN requiring $n$ or more arguments on the stack. If the stack contains at least $n$ elements, then we can

$$f^n;\ a_1 \ldots a_m;\ s;\ H \quad \Rightarrow \quad f;\ a_1 \ldots a_m : s;\ H \qquad \text{(PUSH)}$$

$$f;\ a_1 \ldots a_n : s;\ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)] \quad \Rightarrow \quad e[a_1/x_1 \ldots a_n/x_n];\ s;\ H \qquad \text{(FEXACT)}$$

$$f;\ Code\ a_1 \ldots Code\ a_m; s; H[f \mapsto FUN(x_1 \ldots x_n \Rightarrow e)] \quad \Rightarrow \quad p;\ s;\ H[p \mapsto PAP\ (f\ a_1 \ldots a_m)] \qquad \text{(PAP1)}$$
if $m\ >=\ 1$ *and* $peek(s) \neq\ Code\ y$ *and* $p\ fresh$

$$f;\ Code\ a_n + 1 : s;\ H[f \mapsto PAP\ (g\ a_1 \ldots a_n)] \quad \Rightarrow \quad g;\ Code\ a_1 \ldots Code\ a_n : Code\ a_n + 1 : s;\ H \qquad \text{(PENTER)}$$

(a) $u$-MEM $\beta$-reduction rules.

$$f^n;\ a_1 \ldots a_m : s;\ H[f \mapsto FUN(x_1 \ldots x_n \rightarrow e)] \quad \Rightarrow \quad e[a_1/x_1 \ldots a_n/x_n];\ (\bullet\ a_{n+1} \ldots a_m) : s;\ H \qquad \text{(EXACT)}$$

$$f^n;\ a_{t+1} \ldots a_n : s;\ H[f \mapsto CLO((x_1 \ldots x_n \rightarrow e)\ (\dot{}b_1 \ldots b_t))] \quad \Rightarrow \quad e[b_1/x_1 \ldots b_t/x_t,\ a_{t+1}/x_{t+1} \ldots a_n/x_n];\ s;\ H \qquad \text{(CLOEXACT)}$$

$$\text{Closure}((e)\ (b_1 \ldots b_t)); s; H \quad \Rightarrow \quad e;\ s;\ H[p \mapsto CLO((x_1 \ldots x_n \rightarrow e_1)\ (\dot{}b_1 \ldots b_t))] \qquad \text{(CLOINIT)}$$
$p\ fresh$
$e\ reduces\ to\ x_1 \ldots x_n \rightarrow e_1$

(b) $s$-MSM $\beta$-reduction rules.

Figure 4.2: $\beta$-reduction rules

proceed to evaluate the body of the function, binding the actual arguments to the formal parameters as usual, and leaving any excess arguments on the stack to be (presumably) consumed by the function returned.

- PAP1 generates a runtime partial application if there are not enough arguments on the stack. The rule generates a PAP, which is returned to the caller.

- PENTER is required when we attempt to evaluate a PAP. In this case, we simply unpack the PAP's arguments onto the stack and enter the function.

Consider the application:

```
f  = x -> y -> z -> x + y + z;
f2 = let g = f 2 3;
     in  g 4;
```

To evaluate `g 4`, we first reduce the expression bound to `g` by application of the rule PAP1. This produces a PAP, which we apply to 4 using the rule PENTER.

**Definition 4.3.** *s-MSM $\beta$ reduction rules are defined in Figure 4.2(b).*

- $\text{EXACT}_n$ is used when we are applying the function to n number of arguments.

- CLOEXACT is required when we are applying a Closure to the pending number of arguments.

- CLOINIT is required when we allocate a Closure on the heap which encapsulates an expression that can be immediately reduced. Recall that an encapsulated expression can consist of some number of total applications. These are reduced eagerly when the Closure is created.

The rule CLOINIT is required in the following artificial situation:

```
f  = x -> y -> let h = z -> t -> x + y + z + t; in h;
f2 = let g = f 2 3 4;
     in  g 5;
```

This can be rewritten as

```
f  = x -> y -> let h = z -> t -> x + y + z + t; in h;
f2 = let g = Closure (f 2 3) {4} 1 2;
     in  g 5;
```

when partial applications are statically identified and rewritten during the lambda doping transformatio, which is discussed in chapter 6.

The expression `f 2 3` is encapsulated in the Closure. In this case, when the Closure is allocated and bound to `g`, we reduce the Closure via CLOINIT. This makes the application `f 2 3` the current expression, and it reduces via $\text{EXACT}_n$. A Closure never encapsulates an expression that can be reduced, i.e., is saturated. We use encap. as an abbreviation for encapsulated to ease notation.

It is important to note that both $\text{EXACT}_n$ and CLOEXACT reduce a saturated function call. The only difference for CLOEXACT is that we have some number of presupplied arguments which are included in the reduction step.

## *4.2 The Heap*

Before we discuss implementation issues specific to the MEM and MSM, we look at the way Mondrian compiles heap objects to $C^\sharp$. For the most part this is consistent across both reduction schemes. We adopt the GHC convention and distinguish between different heap objects based on whether they are of *pointed* or *unpointed* type. The pointed and unpointed convention is adopted from domain theory and reflects the fact that computation of such objects may not terminate. Pointed objects include THUNKS, CON and FUN objects, while unpointed objects include mutable arrays and variables (currently not supported).

In the MEM, all executable objects are subclassed from an executable interface object called `Code`, which implements the function `ENTER`. It uses an evaluation mechanism based on a trampoline, a standard reduction technique.

The MSM uses the system stack and the entry point `ENTER` is uniquely determined by each function's type.

### *4.2.1 FUN*

We compile functions to classes. Classes provide a natural way of encoding functions as they provide a way of scoping variables and creating function "instances" which can be manipulated as first class values. We discuss first class functions in more detail when we discuss how functions are compiled on the $\text{CLR}_{\leq}$ and $\text{CLR}_{\leq,\forall}$.

We evaluate all functions in a consistent way by entering a method `ENTER`. Where the function is bound to a string identifier, for example, a `let` binding, we use the `let` binder to name the class. Otherwise, we generate a unique name. For example, given the following:

```
f : Integer -> Integer -> Integer;
f = x -> let g = y -> z -> x + y + z;
          in  g 1 2;
```

This is translated to:

```
class f : Code {

    class g {
        // Instance references to the environment of g.

        Integer ENTER (VMEngine vm)
        {   VM.COLLECT(2, this);  // runtime argument check
            x = VM.POP();
            y = VM.POP();
            ...
        }
    }

    Integer ENTER (VMEngine vm)
    {   VM.COLLECT(1, this);

        // generate an instance of g
        x = VM.POP();
        ...
    }
};
```

Listing 4.2: FUN - $u$-MEM

In push/enter, we use `VM.POP` to pop arguments from the MEM's stack. We will discuss the runtime argument check `VM.COLLECT(2)` in more detail in section 4.3.

In the MSM, the function `f` is compiled to the following:

```
class f : Code {

    class g {
```

```
        Integer ENTER (Integer x, Integer y)
        {   ...;
        }
    }

    Integer ENTER (Integer x)
    {   // generate an instance of g

        ...
    }
};
```

Listing 4.3: FUN - *s*-MSM

Note that the method definition for `ENTER` depends on the type of the function. This is different from the MEM, which is consistent across functions of different type. We make a distinction between a function's type and its *implementation type*. In the case of `f`, its type is:

$Integer \rightarrow Integer \rightarrow Integer$

Its implementation type is `f`. By implementation type we mean the name bound to its `class` definition. The implementation type of a function bound to a let binding will be the same as the name assigned to the `let` binding. The implementation type of an anonymous lambda function will be a uniquely generated name.

We also define two additional entry points: `Apply` and `Eval`. These entry points allow other languages to evaluate a Mondrian expression. Invoking a function via `Apply` will generate an instance of the MEM, which is used to reduce the expression. In contrast, the function `Eval` expects an instance of MEM along with the function's arguments.

### *4.2.2 CLO*

We generate a `class` to represent a Closure. This gives us a convenient place to store the Closure's encapsulated expressions. For example in the following

```
f1 = Integer -> Integer -> Integer;
f2 = map (a -> Closure f1 {a} 1 2) [2, 3];
```

the Closure in function `f2` can be compiled to:

```
class _f1_
```

```
{
    Integer a;

    _f1_ (Integer a)
    {   this.a = a;
    }

    Integer ENTER (Integer b)
    {   f1.ENTER(a, b);
    }
}
```

Listing 4.4: f1 translation to $C^\sharp$

We assume the unique name `_f1_` is generated for the `class` definition. Notice the encapsulated variable `a` is now a field of type *Integer* in the `class` definition. When we evaluate the `Closure`, we have to supply the remaining argument `b`. In the next example, we show how we encapsulate an expression that has a valid application, in the MSM, of $\text{EXACT}_n$.

```
f3 = x -> y -> let z = x + y;
               in  (a -> b -> z + a + b);

main = let f4 = f3 2 3 2;
           f5 = f4 3;
       in  putStr (show f3);
```

The expression `f3 2 3` can be reduced by an application of $\text{EXACT}_n$. This returns the function `(a -> b -> z + a + b)`. The Closure will encapsulate this as the result of the application. This is a function of type

$Integer \rightarrow Integer \rightarrow Integer$

to which we are supplying a single argument `2`. Thus, the type of the whole expression `f3 2 3 2` is

$Integer \rightarrow Integer$

which is the type of the Closure's `ENTER` function. It expects a remaining argument of type *Integer*. Thus, the Closure compiles to the following using $C^\sharp$:

```
class f4 {

    // Cache expression
    F_2 _f;
```

```
    // Buffer encapsulated fields
    Integer a_ = 2;

    f4 ()
    {   this._f = f.ENTER(2, 3);
    }

    Integer ENTER (Integer x)
    {   _f.ENTER(a_, x);
    }
};
```

Listing 4.5: `f4` translation to $C^\sharp$

When we apply the Closure to its remaining argument, we use the encapsulated expression bound to the field `_f`. Note that the type of `_f` is given as `F_1`. `F_1` is a delegate which we use to store function references. It is declared as:

```
delegate Integer F_2 (Integer x, Integer y);
```

We discuss this encoding in more detail in section 7.7.2. Recall that the expression `f 2 3` will be reduced by an application of rule CLOINIT. This reduction now occurs automatically when the Closure is constructed, by putting the reduction in its constructor.

This is a rather simple example, but it shows the essence of our preservation of eager evaluation. Note that in this particular case we can simply insert the literal directly into the expression and save ourselves the extra field.

The preservation of evaluation order is important in the presence of side effects. This is less of a problem in Haskell and Mondrian because unlike $F^\sharp$ and SML.NET we do not support the use of impure constructs. All IO takes place through the threading of the "world" value.

### *4.2.3 CON*

Constructors are discussed in detail in section 2.1.3. We use the `switch` expression to project from the datatype.

### *4.2.4 PAP*

PAPs are used in the MEM and serve much the same purpose as a CLO. They are used to encapsulate functions and state. A PAP, like a FUN, is executable,

and therefore derives from the `Code` class. It is defined by the `PartialApplication` class.

```
public class PartialApplication : Code
{
    internal Vector arguments;
    internal Code code;
    internal int pending;
    internal int total;

    public PartialApplication (Code code, Vector arguments, int pending)
    {
        // ...Store the supplied arguments.
        this.total = arguments.capacity() + pending;
    }

    public object ENTER (VMEngine VM)
    {
        // Push the supplied arguments on the argument stack
        // Try to retrieve the pending arguments
        //      1. If we can't then generate another PAP exception.
        //      2. If we can, execute the encapsulated expression.
    }
}
```

Listing 4.6: Partial application

A `PartialApplication` calculates its pending and total counts from the FUN it encapsulates when it is created. The encapsulated expression always knows how many arguments it wants, and this information is passed to the `PartialApplication` when it is constructed.

### *4.2.5 THUNK*

Thunks are delayed computations used by the MEM. They are implemented by providing a class `Thunk` that takes a reference to a executable object; i.e., one derived from `Code`. It marks the stack and pushes the computation onto the evaluation stack. Termination is either by catching a `PAP` exception or by the function returning normally. In both cases it unmarks the stack and returns. The Thunk is defined below:

```
public class Thunk : Code
{   public object code;
    Boolean virgin = true;
    public Thunk (Code code) {   this.code = code; }
    public object ENTER (VMEngine VM)
```

```
    {   if (virgin)
        {   lock (this)
 8          {   if (virgin)
                {   int old = VM.INSTALL();

                    try
12                  {   object whnf = code;
                        code = null;
                        while (whnf is Code)
                        {   whnf = ((Code)whnf).ENTER(VM);
16                      }

                        VM.RESTORE(old);
                        code = whnf;
20                      virgin = false;
                        return whnf;
                    }
                    catch (PAP e)
24                  {   PartialApplication pap = e.code;
                        VM.RESTORE(old);
                        code = pap;
                        virgin = false;
28                      return pap.code;
                    }
                    catch (DONE_EXEC de)
                    {   ...
32                  }
                }
            }
        }
36      return code;
    }
```

Listing 4.7: Thunk on the $u$-MEM

The code for handling the exception `DONE_EXEC` is omitted, since it is similar to that for handling the `PAP` exception. The stack mark occurs at line 9. After evaluation of the buffered expression, `VM.RESTORE` on lines 18 and 25 is called, supplying the old stack mark stored in variable `old` to restore the stack to its previous state.

Failing an argument satisfaction check results in the exception `PAP` being thrown and captured with the statement `catch(PAP e)`. The extra arguments required are calculated and a PAP heap object created.

## *4.3 Overview of Evaluation in the u-MEM*

This is an overview of evaluation in the Unsaturated Mondrian Exceptional Machine, or $u$-MEM, using the reduction semantics defined in Definition 4.2. The $u$-MEM uses a push/enter method of reduction and abstracts the system stack by building a separate machine running on the CLI. An evaluation loop called a trampoline is used to evaluate an expression to WHNF. For every $\beta$-reduction step, a runtime argument check is performed by the MEM, and, if appropriate, a `PartialApplication` object is constructed to encapsulate the computation. The evaluation loop is given below:

```
public object WHNF (object code)
{
    try
    {   object whnf = code;
        while(whnf is Code)
        {   whnf = ((Code)whnf).ENTER(this);
        }
        return whnf;
    }
    catch (PAP e)
    {
        PartialApplication pap = e.code;
        int extra = pap.arguments.size();
        arguments.setSize(arguments.size() - extra);
        return pap;
    }
    catch (DONE_EXEC se)
    {
        return se.result;
    }
}
```

Listing 4.8: $u$-MEM Evaluation loop

We attempt to reduce the value `whnf` until it is either in WHNF, that is, not a object derived from `Code`, or until we exit the evaluation loop by catching the

exceptions PAP or DONE_EXEC.

PAP exceptions are thrown by a failed runtime argument check. The check is part of a FUN object which we showed in Listing 4.2. The check is reproduced as:

```
public void COLLECT (int number, Code code)
{
    int n;
    if ((n = argumentsOnStack()) < number)
    {   PartialApplication pap
            = new PartialApplication (code,
                                      last(n, arguments),
                                      number - n);
        throw new PAP(pap);
    }
}
```

Listing 4.9: Runtime argument check

The variable number is the number of expected arguments and code the function to encapsulate inside the new PartialApplication. In summary, the runtime argument checking machinery resolves to the following (in no particular order):

1. THUNK marks the stack before evaluating its argument
2. THUNK catches a PAP exception, returning a new PartialApplication object
3. FUN performs a VM.COLLECT(x)
4. WHNF reduction loop catches a PAP exception, returning a new PartialApplication object.

### *4.3.1 Evaluating a "Partial Application"*

We give a simple example, stepping through a reduction that throws a PAP exception.

```
pap1 = let f = a -> b -> a + b;
       in  f 1 2;
```

The lambda expression bound to f is compiled to a THUNK. When the THUNK is evaluated, the stack is marked with the current depth of the arg stack, in this case, 2. Thus the stack looks like Figure 4.3.

When the THUNK evaluates the lambda, it checks for 2 arguments, which fails, and subsequently aborts the evaluation by throwing a PAP exception. While this

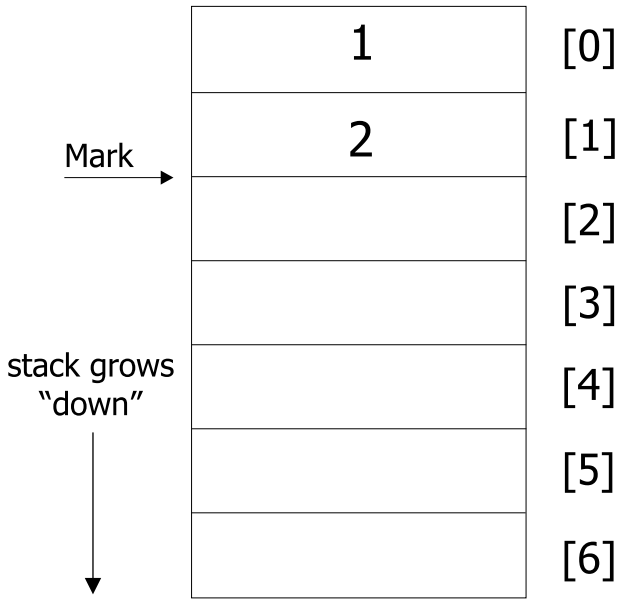


Figure 4.3: Marked stack

is not *strictly* a partial application, because we are not applying `a -> b -> a + b` to anything in the binding `f`, it gives us a useful way to abort the computation at this point. This reduction occurs as a result of PAP1.

The THUNK catches the `PAP` exception and returns a new `PartialApplication` object. It creates this dynamically by caching the lambda bound by `f`, and setting the pending argument count of the `PartialApplication` object to 2. When we leave the THUNK, by returning the new PartialApplication. We remove the stack mark, which resets the mark back to its previous value. Subsequent evaluation of the expression `f 1 2` via PENTER will execute normally.

# Chapter V

# Reduction without Runtime Argument Checking

We will discuss the feasibility of a transformation to convert unsaturated reduction to saturated reduction based on the MSM model of reduction.

In the case of the $u$-MEM, "saturation" is a transformation that results in a program which reduces via a sequence of reductions and does not require the application of the rule PAP1.

Likewise for the MSM, saturation is a transformation that results in a program that does not require a rule for runtime partial applications. However, the situation is a little more complicated due to the location of the runtime argument check. In this case, it is done by the caller. Consider the trivial example:

```
f6  = x -> let g = y -> y; in g;
f7  = f6 2 3;
```

How does the function `f7` know how to build the function call to `f6` without runtime argument checking? Specifically, how many applications of the rule $\text{EXACT}_n$ do we use? In this case, we use two applications of $\text{EXACT}_n$.

This situation does not arise in the MEM because it uses push/enter evaluation semantics where the location of the runtime argument check is within the callee. The callee always knows how many arguments it requires. If all reduction is guaranteed to be saturated, then reduction resolves to two applications of FEXACT.

Thus, in the case of the MSM, we first need a way of faithfully encoding the reduction semantics, so that we can build the correct reduction sequence statically, for all known and unknown function calls. Once we know exactly how a function reduces, we can look at saturating all function calls.

## *5.1 Encoding the Reduction Sequence*

We will look at ways in which we can statically determine the correct reduction sequence for every application in the MSM.

Consider again the example we gave earlier for the functions `f6` and `f7`. The application of `2` and `3` to `f6` would consist of two applications of rule $\text{EXACT}_n$. This first application applies the argument `2` to `f6`. The second application applies the remaining argument to the result of the first application. Two ways naturally suggest themselves to encode this information:

1. Annotating the type information in some fashion to encode the correct sequence of reduction steps,

2. Transforming the program so that we represent applications of $\text{EXACT}_n$ using tupled arguments.

### *5.1.1 Type Annotations*

Type annotations of various forms are an active area of research. They are used in a number of situations, including flow analysis (Faxn 1996*b*), strictness analysis (Jones et al. 2004) and usage analysis (Wansbrough & Peyton Jones 1999). In the case of usage analysis, Jones et al. starts with a type inferred using a derivation of Milner's $\mathcal{W}$ inference algorithm (Damas & Milner 1982), which we call the ML type. From here, they annotate the type with information about how many times each value is used. The evaluation of a value that is only used once can be simplified because we no longer need to update its thunk.

Likewise, we wish to annotate the ML type with information about the reduction of the function. We call these annotations *operational types* or *o*-types. An *o*-type allows us to determine how many times to apply $\text{EXACT}_n$. We will use the type of `f6`, defined in the previous section, as an example. The ML type is given as

$$\forall\alpha\beta.\alpha \to \beta \to \beta \qquad (1)$$

and its *o*-type is given as

$$\forall\alpha\beta.\alpha \to \{\beta, \beta\} \qquad (2)$$

The difference between the types is the use of the annotation $\{\}$, grouping the type variables $\beta$ and $\beta$ to denote a new application of a reduction rule. It is important to realize that $\{\beta, \beta\}$ is strictly an operational annotation. It does not change the type of an expression, `f6` will still take two arguments, typed as $\alpha$ and $\beta$ and return a value of type $\beta$. It allows us to construct the correct reduction semantics given an expression of this type. That is, the reduction of a value typed as the former is a single application of $\text{EXACT}_n$. In the latter, it is two

applications of EXACT$_n$. In this way, we can see that the new type gives us a *direct operational equivalence* between the reduction sequence and the type. Note that we could also encode the ML type as:

$\forall\alpha\beta.\{\alpha,\beta\} \to \beta \qquad (3)$

This is the type of a value that is reduced via an application of the rule EXACT$_n$. Notice the difference between this type and type (2). The annotation $\{\}$ in type (3) does not appear at the "end" of the type. Only operational annotations appearing at the end of a type affect how the value is reduced.

In the case of a value of *o*-type $\alpha \to \{\beta,\beta\}$, the application of EXACT$_n$ returns a value of type $\{\beta,\beta\}$, which is the call continuation. This is really a function of type $\beta \to \beta$. We call the removal of top level $\{\beta,\beta\}$ annotations "unwinding". We must unwind the annotated type to continue the reduction of the call. Consider a more complicated type:

$\forall\alpha\beta.\alpha \to \{\beta,\{\beta,\beta\}\}$

Reducing a value of this type constitutes two applications of rule EXACT$_n$. For every application of the rule EXACT$_n$, we unwind the type once. It is interesting to note that the "equivalent" ML type recovered by unwinding the type twice is:

$\forall\alpha\beta.\alpha \to \beta \to \beta \to \beta$

We can see this type gives us no operational intuition about how a value reduces, typed this way. We can generate *o*-types during inference. The algorithm is discussed later, but for the moment, we assume that we have a type inference algorithm that assigns *o*-types to M$_{CM}$ terms.

The arity of an *o*-type is called its *operational arity*, as opposed to its ML arity, which is the arity of the equivalent ML type. It is possible for a type to have different operational and ML arities. Consider the ML type of `f6`. Its operational arity is 1, while its ML arity is 2. Before we continue, we will look at a slightly more complicated example:

```
foo    = c -> let foo2 = a -> b -> a;
              in  foo2;

main
 = let x  = foo 2 (e -> w -> e + w) 3 2 3;
       x2 = foo 2 foo 3 2 4 3;
   in  putStr (show x);
       putStr (show x2 2);
```

The $o$-type given for the type of the expression

```
foo 2 (e -> w -> e + w) 3 2 3
```

is given as:

$Integer \rightarrow \{\{Integer, Integer, Integer\}, Integer, \{Integer, Integer, Integer\}\}$

This type looks a little complicated, but we can break it down using the MSM reduction rules. The first part of the call `foo 2` is evaluated using $\text{EXACT}_n$, the call continuation represented by the type:

$\{\{Integer, Integer, Integer\}, Integer, \{Integer, Integer, Integer\}\}$

This is the $o$-type assigned to the expression `a -> b -> a`, bound to `foo2` inside `foo`. At this point we "unroll" the $o$-type of the call continuation, represented by $\{\}$. Unrolling a $o$-type constitutes folding the $\rightarrow$ operator along the list of types within the outermost $\{\}$ annotation. We use a function called `expandotype` which unfolds $o$-types (which we will define later). `expandotype` will not unroll an $o$-type if its outermost constructor is a $\rightarrow$ operator.

In general, we unroll the type whenever we cannot apply one of the reduction rules. The type is unrolled to:

$\{Integer, Integer, Integer\} \rightarrow Integer \rightarrow \{Integer, Integer, Integer\}$

We can then apply the rule $\text{EXACT}_n$ which generates the call:

```
(foo 2) ((a -> b -> a + b) 3)
```

We now have the $o$-type $\{Integer, Integer, Integer\}$ remaining. We cannot apply any of the reduction rules, so we unroll the $o$-type, giving us $Integer \rightarrow Integer \rightarrow Integer$. We now use $\text{EXACT}_n$, completing the application, as there are no more call continuations, giving us:

```
(foo 2) ((a -> b -> a + b) 3) (2 3)
```

In the intermediate language $C\sharp$, the call would look similar to this:

```
foo.ENTER(2).ENTER((a -> b -> a + b) 3).ENTER(2, 3)
```

Notice that we have used an `ENTER` wherever we used an application of $\text{EXACT}_n$. For the moment, we do not "compile" the lambda expression, we discuss how this is compiled later. From now on, we will often use the alternative encoding using $C\sharp$ syntax, as it seems clearer. We use `ENTER` to stand for a reduction.

It is important not to "over" roll or "under" roll a type. An "under" rolled type gives an incorrect reduction. For our example, we will use the expression `foo 2 ( a -> b -> a + b ) 3 2 3` again. If we inferred an underrolled $o$-type of

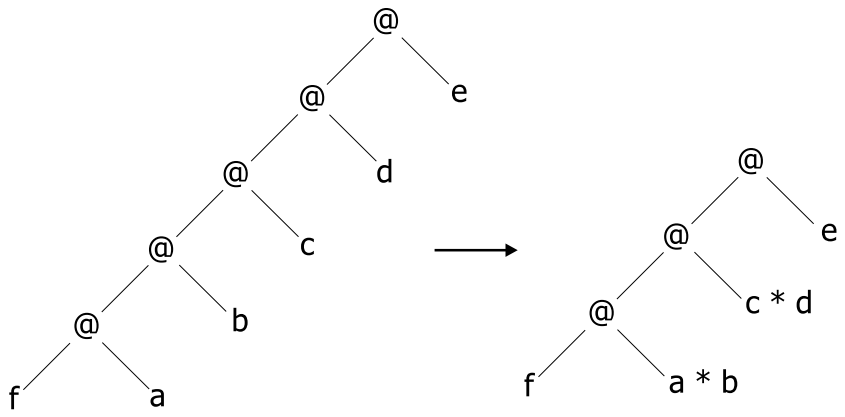


Figure 5.1: The NAry transformation

$Integer \rightarrow \{Integer, Integer, Integer\} \rightarrow Integer \rightarrow \{Integer, Integer, Integer\}$

then this is reduced by EXACT$_n$, producing the reduction `(foo 2 (a -> b -> a + b) 3)`, which is incorrect. Overrolling is not as "bad", because we will unroll until we get an *o*-type that matches a reduction rule. This will usually only ever take a single unroll.

$\{Integer, \{Integer, Integer, Integer, \}, Integer, \{Integer, Integer, Integer\}\}$

This type will require a single unroll before the rule EXACT$_n$ applies. While a value reduced via this *o*-type will be valid, it is not a desired situation, because it introduces more unrolling than required.

### *5.1.2 Transforming the Program*

This method transforms the program to reflect the operational semantics. Instead of using {} to annotate a call continuation, we group the arguments into tuples, which indicate to how many arguments the rule EXACT$_n$ should be applied.

For example, we assume that we have the following call: `f a b c d e`. We then bracket it to reflect the positions of EXACT$_n$ reductions, giving us `(f a b) (c d) e`, which is shown in Figure 5.1.

The arguments `(a b)` and `(c d)` are stored in a parse tree node `AppList`, a tuple, which is designed to mimic the EXACT$_n$ reduction.

Unfortunately, this transformation depends on the generation of more reduction information than we have without using *o*-types. Consider the example we gave in Listing 5.1.1. The reduction of expression `foo 2 (e -> w -> e + w) 3 2 3`

relies on the use of *o*-types to record the locations of call continuations. While we can also directly represent these in the parse tree for the program, we still need a way of generating this information in the first place.

It makes little sense to produce the *o*-type information and then transform the parse tree, when we can read the reduction semantics directly from the *o*-type.

### *5.1.3 Reduction Annotations Summary*

We have described a method for annotating the type information to produce *o*-types, giving us a way to encode reduction in a saturated machine. We have shown that for the MSM we can construct a function call by using the *o*-type inferred for it.

Now, we need a way of saturating all applications. We will look at a simple example and then extend it by case analysis over the terms of the Core Mondrian language, $\mathrm{M}_{CM}$.

## *5.2 Known and Unknown Partial Applications*

A partial application is the result of an unsaturated function call. We will look at examples of partial applications in the context of a known and unknown function.

### *5.2.1 Known Partial Application*

Consider the simple partial application:

```
f  = x -> y -> let g = z -> z; in g;
f2 = f 2;
```

Listing 5.1: Known partial application

This example shows a situation where we have a *known* partial application, in the sense that we can determine statically that the function call `f 2` is not saturated. This expression, in the MEM, reduces by an application of the rule PAP1. However, we can wrap the expression in a Closure, producing the following:

```
f2 = (Closure (f) {2} 1 2)
```

This expression does not match any reduction rule, and evaluation halts. The expression `(Closure (f) 1 2)` is used to encapulsate the function application until the remaining arguments are supplied. In this case, we only require a single argument for the application to be totally applied. We only require a single argument because the o-arity of `f` is `2` and we have already supplied a single argument `2`.

This is different to its ML arity, which is `3`, and gives us the impression that `f` is partially applied. From this simple example, we can see that *o*-types provide a useful way of detecting known partial applications, because *o*-types give us a faithful model of reduction.

### *5.2.2 Unknown Partial Application*

An *unknown* partial application occurs in a situation where we cannot statically determine the exact calling reduction semantics of the function. We discussed this situation in example Listing 1.1, which we include again for reference below:

```
zipWith : (a -> b -> c) -> [a] -> [b] -> [c]
zipWith k [] []         = [];
zipWith k (x:xs) (y:ys) = k x y :: zipWith xs ys;
```

In this function, `k x y` is an unknown application. The reduction semantics of the function `k` is not known until runtime. How then do we statically work out the reduction of `k`? Consider the following function call:

```
f3 = x -> let g = z -> z; in g;
f4 = zipWith f3 [1, 2, 3] [4, 5, 6]
```

Listing 5.2: Unknown `zipWith` application

`f3` is reduced via two applications of $\text{EXACT}_n$. It is apparent that we have a reduction conflict between its use within `zipWith` and its definition. However, `zipWith` indicates statically (by the inferred *o*-type) how it intends to reduce the function. More specifically, the *o*-type inferred for `k` is $(a \to b \to c)$. This is referred to as its *usage* type.

From this information, we can build a function around `f3` which expects to be reduced by a single application of $\text{EXACT}_n$. The *o*-type $(a \to b \to c)$ has thus established a operational contract, to which any function must adhere. We now build the following call:

```
f4 = zipWith (a -> b -> f3 a b) [1, 2, 3] [4, 5, 6]
```

The lambda `(a -> b -> f3 a b)` expects to be reduced via a single application of $\text{EXACT}_n$. The expression `f3 a b` expects to be reduced via two applications of $\text{EXACT}_n$. The addition of the lambda is called a *dope* and the process is called *doping*. The dope has effectively created an operational coercion from one set of reduction rules to another. This marks the transition from an *unknown* application to a *known* reduction.

The detection of this disparity between reduction rules is easily facilitated by simply comparing the o-arity of the supplied argument and formal argument. In this case, there is a mismatch. `f3`'s o-arity is 1 and `zipWith` expects a function of o-arity 2. We sometimes use the term "impedance mismatch", as this usage is consistent with the literature (Jones 1996) and is used for a similar idea.

We can make a further point with this example. If we are reducing via push/enter, the dope is not required. As we remarked earlier, this is a result of the location of the runtime argument check. Intuitively, we do not require the lambda wrap, because `f3` knows how to reduce itself. This shows immediately that push/enter may be a more efficient reduction mechanism, because we will require less dopes. Why then do we still need to bother with eval/apply? Even without runtime argument checking, an implementation of push/enter still requires direct access to its argument stack to push and pop arguments explicitly. This observation allows us to relax the condition we test to detect a reduction disparity for push/enter. In the example, the ML arity of `f3` is the same as that of the function expected by `zipWith`, and therefore, we do not need the dope. We can stop here because `f3 a b` is a total application.

In the following example, the o-arity of the function `(x -> y -> z -> f3 x y z)` is 3, as is its ML arity:

```
zipWith (x -> y -> z -> f3 x y z) [1, 2, 3] [4, 5, 6]
```

Listing 5.3: Doping with zipWith

Doping will produce:

```
zipWith (r -> s -> (x -> y -> z -> f3 x y z) r s) [1, 2, 3] [4, 5, 6]
```

We can treat the dope as in the following expression:

```
(r -> s -> (let g = t -> (x -> y -> z -> f3 x y z) r s t;) in g)
```

This will be lambda lifted to build a Closure

```
(r -> s ->
    (let f = Closure "f" (t -> (x -> y -> z -> f3 x y z) r s t)
                {r, s} 1 1;)
     in f);
```

where `r` and `s` are free in the `let` expression. Alternatively, we can observe that the application `(x -> y -> z -> f3 x y z) r s` is a partial application. We can then encap. the expression directly in a Closure, producing:

```
(r -> s -> Closure "f" ((x -> y -> z -> f3 x y z) r s)
                 {r, s} 1 3))
```

The way we do this, in this example, is immaterial, since the result is functionally equivalent. Both Closure expressions will compile effectively to a function, with `r` and `s` as free variables and expecting a single argument. The numbers `1` and `3` are the pending and total argument counts, respectively. Reduction halts at the Closure because there are no reduction rules that apply until the remaining argument is supplied. For example, in the following,

```
let x  = zipWith (r s -> (Closure (x y z -> x y z) {r, s} 1 3))
                         [1, 2, 3] [4, 5, 6];
    x2 = map (a -> a 2) x;
in  ...
```

mapping the function `(\a -> a 2)` over the list `x` will invoke three applications of CLOEXACT. The reader may have noticed that in this case we could have rewritten Listing 5.3 as the following:

```
zipWith f [1, 2, 3] [4, 5, 6]
```

This is because the operational arity of `f` is the same as the arity of the function required by `zipWith`. Thus, in some special cases like these, we can dope by *removing* the lambda. This is more of an artificial example, but when it does occur (which would be rare in a real world application) it is useful to know that it can be removed without any problems.

We are now in a position to define where unknown application occur, which is where we attempt to use a function of one o-arity where another of a different o-arity was expected. Intuitively this occurs in two places only. The first place is where the *o*-type of a formal argument is different from the supplied argument, and the second place is in a sum type, where the supplied function is of a different *o*-type, from the ML type specified by the field definition. Recall that sum types are defined by supplying the ML type of the field. In both locations, we have separated the definition of the function from its application via an argument or field that may be typed with a different *o*-type (function) or ML type (sum). We discuss constructing sum types in detail in section 5.4. This separation in $\mathrm{M}_{CM}$ does not occur in any other location.

### *5.2.3 Summary*

In summary, we have illustrated how we have used *o*-types to model reduction semantics. We have shown in both known and unknown situations, that *o*-types allow us to determine if a reduction results in an partial application. Listing 5.1 demonstrates the former, while Listing 5.2 demonstrates the latter.

We showed that detection of a reduction mismatch in an unknown location using eval/apply could be detected using the o-arity of the supplied and formal arguments. We showed that when using push/enter the reduction mismatch can be detected by using the arity of the ML types. We then showed that we can resolve the impedance mismatch by using an operational coercion. The coercion produces a dope or lambda wrap, which allows us to reduce the expression using the reduction semantics of the caller without requiring runtime argument checks.

Our next step is to show, by case analysis of the different expressions, that we can produce valid operational coercions using the expressions' *o*-type to its usage type, that is, its formal argument *o*-type or sum field ML type.

## *5.3 Doping Case Analysis by example*

Our treatment of case analysis is by example over the $M_{CM}$ terms. While it is less formal, it suffices to show that we can produce operational coercions for every type of expression passed to a unknown application site.

### *5.3.1 Variable*

This case is trivial. We use the *o*-type of the variable to build a lambda, if there is an impedance mismatch.

### *5.3.2 Lambda*

This case is trivial. An operational coercion is built using the *o*-type of the lambda, if there is a impedance mismatch.

### *5.3.3 Conditional*

Conditional expressions are complicated by the fact that they may return functions which reduce via a different set of $\text{EXACT}_n$ reductions. For this discussion, we use the following functions:

```
boolFlag1 = True;
boolFlag2 = False;

//func1 : Integer -> {Integer, Integer};
func1 : Integer -> Integer -> Integer;
func1 = x -> plus x;

// func2 : Integer -> Integer -> Integer
func2 : Integer -> Integer -> Integer;
func2 = x -> y -> x + y;

// func3 : {Integer, Integer} -> Integer -> Integer;
func3 : (Integer -> Integer) -> Integer -> Integer;
func3 = f -> a -> f a;

// func5 : Integer -> {Integer, Integer}
func5 : Integer -> Integer -> Integer;
func5 = x -> plus x;

// func4 : {Integer, Integer} -> {Integer, Integer}
func4 : (Integer -> Integer) -> Integer -> Integer;
func4 = f -> func5 (f 2);
```

Listing 5.4: Sample functions

We will first look at where an `if` condition appears as an argument in an application. We treat each branch of the conditional separately and generate a dope coercion where appropriate.

```
main
 = let  // x1 : [Integer -> Integer];
        x1 = map (if (boolFlag1) func1 else func2) [1, 2, 3];

        // x2 : [Integer];
        x2 = foldr (if (boolFlag2) func3 else func4) 2 x1;
   in   putStr (show x2);
```

Listing 5.5: `if` as an application argument

This example illustrates a number of points. Consider the first application:

```
map (if (boolFlag1) func1 else func2) [1, 2, 3]
```

In this case, both `func1` and `func2` may be used inside `map`, depending on the value of the scrutinee. Their ML types are the same, that is, $Integer \rightarrow$

*Integer* → *Integer*, but their *o*-types differ. Since although both functions are of arity 2, applying a value to `func1` using EXACT$_n$ does not produce a runtime partial application, while doing the same for `func2` results in a partial application.

To get around this, we treat `func1` and `func2` separately, and apply the transformation over the components of the `if` expression. Thus the expressions `x1` and `x2` are translated into:

```
x1 = map if((boolFlag1))
         then func1
         else (a ->
                (Closure (func2) {a} 1 2)) [1, 2, 3]))
```

Listing 5.6: `x1` doped

Notice that we have not doped the function `func1`. This is because its *o*-type is compatible with that expected by `map`. That is, it will reduce with a single application of EXACT$_n$. Note the presence of `x2` and `x1` in the Closure. These are free variables, which are part of the environment of the call. In the second case

```
foldr (if (boolFlag2) func3 else func4) 2 x1
```

both `func3` and `func4` are of the same ML type, but we only dope `func4`. This is because the calling semantics of `func4` are different from those used by `foldr`. We use two applications of EXACT$_n$ to call `func4`, while inside `foldr` we use one application of EXACT$_n$, supplying two arguments, which is incorrect. We therefore produce the following:

```
x2 = (Closure "x2"
         (foldr (if (boolFlag2)
                then func3
                else (a -> b -> func4 a b)) 2 x1) {x1} 0 0)
```

Listing 5.7: `x2` doped

Recall that `x1` is considered free in the expression and is included in the Closure's free environment. In the case of push/enter, however, we do not produce the dope.

Consider the following example where the `if` expression occurs as the primary function symbol:

```
f = x -> if (boolFlag1)
         then func1;
         else func2;
```

```
g = f 2 3;
```

Listing 5.8: `if` as a principle function symbol

In this example, we may return either `func1` or `func2`, depending on the value of `boolFlag1`. If we return `func1` then we reduce the expression `f 2 3` via two applications of EXACT$_n$. Otherwise we only require one application of EXACT$_n$.

We can resolve this by selecting one of the conditional expression's *o*-types and using it to establish an operational contract, against which we dope the remaining branch. The only remaining problem is which *o*-type do we pick? In the case of Listing 5.8, we pick the *o*-type of `func1`. If we picked the *o*-type of `func2`, then we would produce the following dope:

```
f = x -> if (boolFlag1)
         then (a -> b -> func1 a b);
         else func2;
```

Listing 5.9: Dope using `func2`'s *o*-type

The *o*-type of `f` is $Integer \rightarrow Integer \rightarrow Integer$ which means that the expression `f 2 3` will reduce via the expression

```
f.ENTER(2, 3);
```

This application results in the correct reduction of both `func1` and `func2`. However, if we modify the application slightly to `f 2`, then we end up creating a Closure of the form

```
g = Closure (f) {2} 1 2
```

This is incorrect because the reduction of `func1` can proceed with a single application of EXACT$_n$. If we choose the *o*-type for `func1` as the dope target, then we will produce the dope

```
f = x -> if (boolFlag1)
         then func1;
         else (a -> Closure func2 {a} 1 2);
```

Listing 5.10: Dope using `func1`'s *o*-type

The type of `f` is $Integer \rightarrow \{Integer, Integer\}$. We have effectively "pushed down" the Closure so that the reduction `f 2` will now proceed if the value of `boolFlag1` is `True`. We can see that the appropriate choice in this case is the expression with an *o*-type that requires the most applications of EXACT$_n$.

In most situations this strategy works. However, there are situations where it does not. This can arise where the $o$-type containing the most number of applications of $\text{EXACT}_n$ will result in an invalid reduction sequence for the other expression. For example, assume we have two expressions $e_1$ and $e_2$ typed respectively with the $o$-types $\alpha \to \{\beta, \phi, \gamma, \tau\}$ and $\alpha \to \beta \to \{\phi, \{\gamma, \tau\}$. The later $o$-type has the most number of applications of $\text{EXACT}_n$. Doping $e_1$ against $e_2$'s $o$-type would result in a dope requiring two applications of $\text{EXACT}_n$ before $e_1$ could be reduced. Thus, the reduction of $e_1$ has been delayed, and additional Closures will be generated where there is an outstanding application of $e_1$ to a single argument. We can modify the $o$-type we dope against by "combining" the two $o$-types into a *sum*. The sum of the two $o$-types will be $\alpha \to \{\beta, \{\phi, \{\gamma, \tau\}\}\}$, which will result in the most optimal reduction sequence for both expressions.

It should be apparent that the only real difference in doping a conditional expression is which $o$-type against which we test the impedance match. Where the conditional is an argument in an application, the target $o$-type is the $o$-type of the function to which the argument is being applied. Where the conditional is the principle function symbol, we dope each conditional expression against the $o$-type of the `if` expression.

### *5.3.4 Switch*

`Switch` expressions are doped in a similar way, and are treated desugared as `if` expressions.

### *5.3.5 Let*

Doping `let` expressions is trivial and is implemented in the obvious way. We simply dope the body of the let expression and then dope each binding separately.

### *5.3.6 Closure*

Since Closures are now part of the $\text{M}_{CM}$ grammar, we have to consider this case. A Closure just stores an encapsulated expression. Therefore, we can treat the encapsulated expression using the case analysis presented thus far.

### *5.4 Doping Sum Types*

Sum types present us with a few complications if we are compiling to the MSM. Consider, for example, the following:

```
class FuncStore {};

class FuncStore1 : FuncStore
{   flist1 : List<Integer -> Integer>;
};

class FuncStore2 : FuncStore
{   func2 : Integer -> Integer -> Integer;
};

class FuncStore3 : FuncStore
{   func3 : Integer -> Integer -> Integer -> Integer;
};

class FuncStore4 : FuncStore
{   func4 : Integer -> Integer -> Integer -> Integer -> Integer;
};
```

There is no way of knowing when we project the field `func3` if the function projected by `func3` is a function of *o*-type

$\{Integer, Integer\} \rightarrow Integer \rightarrow Integer$

or is a function of *o*-type

$\{Integer, Integer\} \rightarrow \{Integer, Integer\}$

This is a problem because doping relies on static information at the call site to build the correct dope and closure environments. For example, given the above sum type, the type inferred for the function `f` assigned to the field `func3` in the application `f 2 3 4` will be:

$Integer \rightarrow Integer \rightarrow Integer \rightarrow Integer$

This will be incorrect, if the actual *o*-type of the function assigned is given as:

$Integer \rightarrow Integer \rightarrow \{Integer, Integer\}$

An easy solution is to treat field bindings as we do function application, by generating a coercion from the bound value to that expected by the field type declaration.

### *5.4.1 Generating Closures on Field Assignment in Non-Generic Sums*

Non-operational sum function types increase the number of dopes produced by the doping transformation under eval/apply. They reduce the efficiency because they introduce "non-deterministic" evaluation in the sense that the user assigned type does not accurately reflect the reduction semantics of the value assigned to the sum. The solution is to generate dopes upon impedance mismatches during sum creation. This is a more practical solution, as it does not reduce the set of functions matching the field type signature. Therefore, it will not reject programs with mismatching operational typings. For this reason, we generate an operational coercion between the sum field type and the function assigned to it. This guarantees that whenever the field is projected, it projects a function that is *operationally* equivalent to the field's ML type. We refer to the *assigned* function as the function stored or *injected* into the field of the sum type, and the *projected* function as that *projected* from a sum type. Both operations are defined in the usual way. For example, using the functions defined in Listing 5.4, injecting the function `testf` into the field `FuncStore2.func2` would generate a doping. Consider the following:

```
func3 = let newdata = FuncStore2 { func2 = testf; };
        in  ...
```

The assignment of the function `testf` generates an impedance mismatch with the field's ML type which causes the dope rewrite to the following:

```
func3
 = let newdata
        = FuncStore2
         { func2 = (a -> b -> testf a b);
         }
   in  ...
```

The Closure expression `testf a b` would then reduce via two applications of $\text{EXACT}_n$ which, in $C^\sharp$, would be:

```
testf.ENTER(a).ENTER(b)
```

In contrast, the lambda dope reduces by a single application of $\text{EXACT}_n$, the call translating to `testf.ENTER(a, b)` in $C^\sharp$. In this example, we will create different instances of `FuncStore3`, by binding different functions:

```
// testf1 : Integer -> Integer -> Integer -> Integer;
// testf2 : Integer -> Integer -> {Integer, Integer};
// testf3 : Integer -> {Integer, {Integer, Integer}};
```

```
// testf4 : Integer -> Integer -> {Integer, {Integer, Integer}}};

main  : IO<Void>;
main  = let newdata  = FuncStore3 { func3 = testf1; };
            newdata2 = FuncStore3 { func3 = testf2; };
            newdata3 = FuncStore3 { func3 = testf3; };
            newdata4 = FuncStore4 { func4 = testf4; };
        in  let x  = func3 newdata;
                x2 = func3 newdata2;
                x3 = func3 newdata3;
                x4 = func3 newdata4;
            in  do
                {   putStr (x -> x 2 3) x;
                    putStr (x -> x 2 3) x2;
                    putStr (x -> x 2 3) x3;
                    putStr (x -> x 2 3) x4;
                };

func3 : FuncStore -> [Integer -> Integer -> Integer];
func3 = x -> switch x of
             {   case FuncStore3 { f = func3; } :
                        map f [1..10];
             ;   case FuncStore4 { f = func4; } :
                        map (f 2) [1..10];
             };
```

Listing 5.11: Different sum bindings

The following dope rewrites occur (note that we annotate the bindings with brackets to show how reduction will occur):

```
main
 = let newdata
        = FuncStore
          { func2 = testf1; };
       newdata2
        = FuncStore2
          { func2 = (a -> b -> c -> (testf2 a b) (c)};
       ...
   in  ...
```

Listing 5.12: Sum dopings

We can omit the rewrite for the function `testf1` because its *o*-type is the same as the fields. We generate the usual rewrite for the map applications. The first one for expression `map f [1..10]` is given by:

```
map (d -> (Closure (a -> b -> c ->
                        testf1 a b c) {d} 2 3))
                            [1..10];
```

Doping guarantees that the lambda "bubble" protecting the expression is of the *o*-type $Integer \rightarrow Integer \rightarrow Integer$ while the doped expressions `testf1 a b c`, `testf2 a b c`, etc., will observe different reduction semantics. We can use the result of `func3` in the usual way. For example,

```
main = let // x : [Integer -> Integer -> Integer]
           x  = func3 (FuncStore3 { func3 = testf3; });
           x1 = map (f -> f 2 3) x;
       in  putList x1;
```

`putList` is defined to print a list of `Integers` as `Strings`. The application `f 2 3`, in this case, will be compiled to

```
f.ENTER(2, 3)
```

which translates to the call:

```
testf3.ENTER(a).ENTER(b).ENTER(c)
```

### *5.4.2 Generic Sums*

Generic sums introduce polymorphism across sum fields by providing the ability to instantiate quantifiers to concrete types. Initially, this suggests that we can ease the lambda doping burden on fields because we can specialize the type annotations to the appropriate *o*-type during inference. This, however, is not the case. Consider the following example:

```
// testf3 : Integer -> Integer -> {Integer, Integer};

main  = let x1 = map testf3 [1..10]
        in  func3 x1

func3 : forall a, b, c => [a -> b -> c] -> a -> b -> [c];
func3 = xs -> switch xs of
              { []   : xs;
              ; x:xs : x a b :: func3 xs a b;
```

```
}
```

This is an instructive example, because it illustrates a problem with packaging functions within generic data structures. `testf3` will be doped because of the `map` impedance match, giving us:

```
(a -> Closure (testf3) {a} 1 2)
```

The `map` application will then create a list of partially applied Closures. If we take no further steps, then the variable `xs` will be a reference to a list of functions of $o$-type $Integer \rightarrow \{Integer, Integer\}$ which, if we supplied to the function `func3` as is, would be incorrect. The problem is that `func3` expects a list of lambdas of $o$-type $Integer \rightarrow Integer \rightarrow Integer$, assuming we instantiate `a, b` and `c` to $Integer$. The inefficient way of resolving this impedance mismatch is to generate code that will walk the list and generate the appropriate dopings based on the callee's argument type signature after the application `map testf3 [1..10]`. This assumes a knowledge of how to "traverse" every defined data type capable of storing functions.

An alternative way is the same way we solved the impedance mismatch for field signatures of non-generic classes. We assume that we will get an impedance match between the `Cons` constructor's `head` field type and the binding's $o$-type, and therefore, we generate a dope. In this example, this means `xs` will now be a list of functions of $o$-type $Integer \rightarrow Integer \rightarrow Integer$ and contain a list of

```
(b -> c -> (a -> Closure (testf3) {a} 1 2) b c)
```

expressions. The encapsulated expression will reduce by the following call sequence, in $C\sharp$:

```
testf3.ENTER(a).ENTER(b).ENTER(c)
```

The lambda, however, will reduce by the following:

```
(lambda).ENTER(a, b, c)
```

This preserves the different calling semantics and provides the appropriate translation. We can demonstrate this more clearly if we consider an example using a data structure that must be projected explicitly using a `switch/case` construct. Consider the following example:

```
class FuncStore4<A> : FuncStore
{   func4 : A;
};
```

If we construct an instance of this type via the following,

```
// testf2 : Integer -> Integer -> {Integer, Integer};
f1 : forall a, b => FuncStore4<a -> a -> b> -> a -> a -> b;
f1 = x -> y -> z ->
             switch x of
             {  case FuncStore { f = func4; }:
                    f y z;
             };

f2 = let newdata = FuncStore4 { func4 = testf2; };
     in  func1 newdata;
```

`newdata`'s quantifiers `a` and `b` will be instantiated as `Integer -> Integer -> Integer` and `{Integer, Integer}`, respectively. The dope translation is the same as that given in Listing 5.12 for `testf2`.

### *5.4.3 Product Doping*

Mondrian in its current form does not explicitly support tuple types as implemented in such languages as Haskell. Therefore, we have not extended the solution presented so far. We discussed in section 2.1.3 how we "modeled" them using sum types by using the following construction:

```
class Pair<t1, t2>
{   a : t1;
    b : t2;
};
```

A tuple of elements `(a, b, c)`, using the function definitions given in section 5.11, would translate in pseudo $C^\sharp$ to:

```
new Pair<Integer, Pair<Integer, Integer>>
      ( a : Integer :
            new Pair<Integer, Integer>
                ( a : Integer, b : Integer))
```

In the case where the tuple is `(func1, func2, func3)`, we would generate the dopes given in section 5.12. We then construct a `Pair` structure given in pseudo Mondrian:

```
Pair { a = (a -> b -> c -> Closure (func1 a b c) {} 0 3));
       b = Pair { a = (a -> b -> c ->
                           Closure (func2 a b c) {} 0 3));
```

```
                b = (a -> b -> c ->
                        Closure (func3 a b c) {} 0 3));
            };
    };
```

### *5.4.4 Doping `Seq` and Polymorphic Function Calls*

We will finish this section by looking at some more complicated examples of doping.

### *5.4.5 Doping the Polymorphic `Seq` Operator*

A classic example in which the scrutinee cannot be statically determined to be a non-function type is the use of the polymorphic `seq` function defined in Core Haskell in the following way:

```
seq :: a -> b -> b;
seq a b = case a of { x -> b }
```

Expression evaluation is forced by using a `case` expression. We can also express this in Mondrian using:

```
seq :: forall a,b => a -> b -> b;
seq a b = switch a of { default: b; };
```

Like Mondrian, GHC does not statically determine if the argument `a` to `seq` is a function and therefore, when compiling to push/enter, does not know if the stack has to be marked. However, in Mondrian we can guarantee that the expression will evaluate without constructing a partial application, or consuming too many arguments off the stack. In the case of GHC, this means we can dispense with the stack mark. Consider the following example, given in Haskell to save clutter, using `seq`:

```
// f1 builds up a list of partially applied functions
f1 = p1 (\x -> \y -> \z -> (x+1, y-1, z*1)) [1..10] [11..20];

p1 : (a -> b -> c) -> [a] -> [b] -> [c];
p1 f (x:xs) (z:zs)
 = let  f2 = f x;
        f3 = f x y;
   in   seq (f2) (seq (f3) (f3 :: p1 xs ys));
```

The function `p1` takes a two argument function, which is applied to a single value `x`. This is a partial application which will be transformed to:

```
Closure "f2" (f2) {x} 1 2
```

After lambda doping, we get the following:

```
f1 = p1 (\x -> y ->
            Closure (\x -> \y -> \z -> (x+1, y-1, z*1)) {x, y} 1 3)
                [1..10] [11..20];

p1 : (a -> b -> c) -> [a] -> [b] -> [c]
p1 f (x:xs) (z:zs)
 = let  f2 = (Closure "f2" (f) {x} 1 2);
        f3 = f x y;
   in   seq (f2) (seq (f3) (f3 :: p1 xs ys));
```

Again, `p1` expects the function `f` to be of arity 2. The partial application to the variable `x` in the expression `f x` is rewritten by doping to the closure `(Closure "f2" (f) {x} 1 2)`. When considering the first `seq` application, `seq` attempts to reduce `f2`, which does not work, as there are no evaluation rules that match reducing a Closure. In contrast, the second application of `seq` will reduce via an application of the rule $\text{EXACT}_n$ returning the Closure:

```
Closure (\x -> \y -> \z -> (x+1, y-1, z*1)) {x, y} 1 3)
```

Note that if "some" evaluation is possible when reducing `f2` to `seq`, then the dope of `f2` will be transformed into something like

```
case (f x) of a -> Closure (a) {} 1 2
```

where we reduce `f x` first and return the rest of the computation. This transformation is discussed in more detail in section 4.3.1.

Lastly, we consider this example (given in Mondrian):

```
f : forall a => a -> [a];
f x -> let y = List { head = x; tail = Nill} in y;

g = switch (f (v -> let f2 = b -> v + b in f2)) of
    {   case Cons { h = head; t = tail} : h 2 3;
    };
```

This example uses generic sums and argument impedance matching. We explained in section 5.4.2 that we generate a dope when we bind functions to fields. How do we know in the case of function `f` if the value bound to variable `x` is a function? We rely on the impedance mismatch generated in the application

```
f (v -> let f2 = b -> v + b in f2)
```

between the caller and callee. This will generate the dope, given as:

```
(c -> d -> (a -> let f2 = b -> a + b in f2) c d)
```

Within the dope, we know how to generate the call to the function, giving us, in pseudo $C^\sharp$

```
(a -> let f2 = b -> a + b in f2).ENTER(c).ENTER(d)
```

while the expression `h 2 3` will be compiled to:

```
h.ENTER(2, 3)
```

## *5.5 Doping Summary*

We have described a system in which correct reduction semantics can be maintained without the need for runtime argument checks. We are now in a position to describe a method for which we can compile to the system stack using the reduction semantics for the MSM.

We will then describe a system for compiling to the system stack using the intermediate language $C^\sharp$. It should be noted that $C^\sharp$ is just syntactic sugar, but it gives us an approachable syntax for describing the translation process. We will next describe the doping algorithm.

# Chapter VI

# Middle : Lambda Doping

The lambda doping transformation consists of two separate transformation phases:

- Encapsulation of partial applications by statically isolating them within a Closure called *pap lifting*. This transformation encapsulates *known* partial applications.
- Application of operational coercions in which we have an impedance mismatch between the supplied argument and formal argument reduction semantics. This transformation is used by the MSM to preserve reduction semantics and encapsulate *unknown* partial applications.

## *6.1 PAP lifting*

This transformation isolates known runtime partial applications within a Closure. A Closure, given an $o$-type, reduces its encapsulated expression via a set of reductions specified by the operational semantics given in section 4.1. This is what we are doing during compile time: reducing the application using the operational semantics, and keeping additional bookkeeping information such as pending and total counts. This lends us a useful method of saturating known partial applications.

### *6.1.1 Closure*

We gave the grammar for a Closure in section 3.1.1; here, we describe the Closure's fields in more detail:

- $c$ : the *name* assigned to the closure. This is useful for named closures, that is, those generated by let bindings
- $e_1$ : the *encapsulated* expression

- *str* : the unique *name source* that generates fresh names by taking into account the encap. expression's lexical context, that is, its free environment, required to avoid name capture
- $\overline{e_i}$ : the list of *applied* arguments. These arguments will be applied, in addition to the pending arguments, when the encapsulated expression, $e_1$, is reduced.
- $v$ : the *pending* number of arguments required by the encapsulated expression.
- $v$ : the *total* number of arguments required by the encapsulated expression. This number includes the number of applied arguments.

**Definition 6.1.** *The operational semantics for closure manipulation are given in Figure 6.1.1. We use* $\Rightarrow$ *to show the mapping of the expression to the result and present the rules using Haskell style pattern matching on the Closure components. We use* $arg_1 \ldots arg_n$ *to delineate the applied arguments,* $farg$ *for an argument free in the expression* $e$*,* $p$ *the pending count and* $t$ *the total applied count. The unique set of random names is denoted as* $NS$*. The set of free arguments and applied arguments that are variables is given by* $CV$*.*

*The function* `oarity` *returns the number of arguments denoted by the o-type of an expression. Adding a variable argument to the Closure reduces the number of available names in the name source* $ns$*. We define the new name source* $ns'$ *as the set* $\{v \mid v \in NS \wedge \notin CV\}$*. The type of the Closure is denoted by the variable* $\sigma$*. We can access the o-type's head and tail using the functions* `typeHd` *and* `typeTl`*. The function* `isFunction` *takes a type and checks if its head constructor is a* $\rightarrow$*. We assume for the rule CL_CLOSE that* $e'$ *is generated by the expression* $foldl(lhe \rightarrow rhe \rightarrow lhe$ '@' $rhe)$ $e$ $(arg_1, \ldots arg_n)$*, where @ is the application operator. Finally, to save space, we use* $Cl$ *to denote a Closure object.*

The rules should be straightforward. The rules CL_ADD1 and CL_ADD2 add expressions to a Closure. If the pending count is not 0, then we can add an expression to the Closure. This decrements the pending count and removes the top-most $\rightarrow$ constructor of the Closure type. The new Closure type is now the tail of the original type. We can also add an argument to a Closure if the pending count is 0 and the Closure's type is a function type. At this point, we have a total application to the encap. expression. We can now apply the encap. arguments to the encap. expression and store the resulting expression as the new encap. expression. In

$$
\begin{array}{rcl}
 & & \text{(CL\_NEW)} \\
c\ e\ ns & \Rightarrow & Cl\ c\ \{e\}\ ns\ \{\}\ (arity\ e)\ (arity\ e) \\
 & & \\
 & & \text{(CL\_ADD1)} \\
Cl\ c\ \{e\}\ ns\ arg_{n+1} : \{arg_1, \ldots, arg_n\}\ p\ t & \Rightarrow & Cl\ c\ \{e\}\ ns'\ \{arg_1, \ldots, arg_{n+1}\}\ (p-1)\ t \\
\quad where\ p\ >\ 0\ and\ p\ <\ t\ and\ t\ >\ 0 & & \sigma'\ =\ typeTl(\sigma) \\
 & & \\
 & & \text{(CL\_ADD2)} \\
Cl\ c\ \{e\}\ ns\ arg_{n+1} : \{arg_1, \ldots, arg_n\}\ p\ t & \Rightarrow & Cl\ c\ ns'\ \{arg_1, \ldots arg_{n+1}\}\ p'\ t' \\
 & & \quad \sigma' \ = \ expandotype(\sigma) \\
\mid p\ ==\ 0\ and\ t\ >\ 0\ and\ isFunction(\sigma) & & \quad t' \ = \ oarity(\sigma') \\
 & & \quad p' \ = \ t-1 \\
 & & \\
 & & \text{(CL\_REMOVE)} \\
Cl\ c\ \{e\}\ ns\ \{arg_1, \ldots, arg_n\}\ p\ t & \Rightarrow & Cl\ c\ \{e\}\ ns'\ \{arg_1, \ldots, arg_{n-1}\}\ (p-1)\ t \\
\quad where\ p\ >\ 0\ and\ t\ >\ 0 & & \\
 & & \\
 & & \text{(CL\_ADDFREE)} \\
Cl\ c\ \{e\}\ ns\ farg_1 : \{arg_1, \ldots, arg_n\}\ p\ t & \Rightarrow & Cl\ c\ \{e\}\ ns\ \{arg_1, \ldots, arg_n, farg_1\}\ p\ t \\
 & & \\
 & & \text{(CL\_REMFREE)} \\
Cl\ c\ \{e\}\ ns\ \{arg_1, \ldots, arg_n, farg_1\}\ p\ t & \Rightarrow & Cl\ c\ \{e\}\ ns\ \{arg_1, \ldots, \mathrm{arg}_n\}\ p\ t \\
 & & \\
 & & \text{(CL\_CLOSE)} \\
Cl\ c\ \{e\}\ ns\ \{arg_1, \ldots, arg_n\}\ p\ t & \Rightarrow & e' \\
\quad \mid p\ ==\ 0 & &
\end{array}
$$

Figure 6.1: Closure reduction rules

this situation, the type is *unrolled*, using `expandotype`, and this type becomes the Closure type. The pending and total argument counts are recalculated from the o-arity of the new type. Thus, in effect, the "current state" of the closure is either a partial application or a total application that stores free environment. This distinction is important when we want to maintain correct evaluation semantics. We do not want to keep total evaluations pending, if there are enough arguments to reduce the expression. We refer to this notion as "eager" caching and an expression that can be reduced within the Closure as an "eager" expression.

Closures store free environment in let bindings, hence the rules for CL_ADDFREE and CL_REMFREE. If the Closure has pending count 0, and we call CL_CLOSE then the Closure is *unwound*. Unwinding a Closure via CL_CLOSE will return an expression formed by applying the accumulated arguments to the encapsulated expression. For example,

```
(Closure (f) {2, 3} 0 2)
```

is trivially unwound to `f 2 3`. Note that a Closure will not unwind itself if it is a total application and is storing free environment, which is why it is used during lambda lifting. We provide a function `unfoldClosure` which will unfold a Closure, disregarding the value of its pending argument count.

The unique name $ns$ source is also recalculated as $ns'$ whenever we add an argument to the Closure. When we compile the Closure, any pending arguments must be applied to the Closure, for each of which we need unique binders. At this point, a useful optimisation is used when we encapsulate anonymous lambda expressions. Lambda expressions already provide unique names for the pending arguments, so the first $n$ elements of the name source list are used as the names of the pending arguments.

We will briefly illustrate the reduction semantics of a Closure, using the example we gave in section 5.1.1. Recall that the $o$-type for the function `foo` in the application

```
foo 2 (e -> w -> e + w) 3 2 3
```

is given as:

$$Integer \to \{\{Integer, Integer, Integer\}, Integer, \{Integer, Integer, Integer\}\}$$

We start with the rule CL_NEW to construct a new Closure, using the principle function symbol `foo`, which gives us the Closure:

```
Closure "x" {foo} {} 1 1
```

Adding the argument `2`, using rule CL_ADD1, results in a total application and decrements the pending argument count to 0. This is equivalent to an application of the rule $\text{EXACT}_n$. We then unwind the type because no reduction rules apply. As we noted earlier, unwinding the type means we are ready to apply another reduction rule. Thus we reset the pending and total argument counts to that indicated by the arity of the $o$-type. We now have the following Closure:

```
Closure "x" {foo 2} {} 2 2
```

This allows us to apply the rule CL_ADD2, which adds the arguments

```
(a -> b -> a + b)
```

and `3`. This process is repeated for the remaining arguments.

### *6.1.2 PAP Lift Transformation*

A pap "lift" is the use of a Closure to encapsulate a partially applied expression in order that no runtime argument checks are required.

**Definition 6.2.** *The algorithm* `paplift` *pattern matches on the expression $e$ of type $\tau$. It is given in Listing 6.1. We build expressions, using the Closure defined in Definition 6.1, to implement the algorithm given by induction over terms in Listing 6.1. Intuitively, we start at the leftmost leaf, or principle function symbol, of an application and assume that it is a partial application. Walking up the application nodes, we systematically add the applied arguments until we reach the last application. At this point, we can decide, using the pending count, if the application was fully applied.*

*We omit the explicit application of a new name and name source in the rule* CL_NEW. *We abuse notation and use CL_ADD$^n$ where $n \in \{1, 2\}$ to stand for rules* CL_ADD1 *or* CL_ADD2. *The appropriate selection is decided by satisfaction of the guards listed in Definition 6.1.*

*We call* CL_CLOSE *after every paplift, which will unwind the Closure if we lack a partial application.*

Note that we have a slight complication when we combine the `paplift` transformation with the lambda lifting. This arises from the following:

```
f = x -> let f1 = f2 x;
             f2 = x -> y -> ....;
         in  ...;
```

```
paplift (e : τ)
    if e = x
         return (CL_NEW(x))
    else if e = (λx.e1)
         let
             e'1 = CL_CLOSE(e1)
         in  return (CL_NEW (λx.e'1))
    else if e = e1 : τ1  e2 : τ2
         let e'2 = CL_CLOSE(paplift e2)
             e'1 = paplift e1
         in  return CL_ADD^n (e'1, e'2)
    else if e == let x = e1 in e2
         let e'1 = CL_CLOSE (paplift e1)
             e'2 = CL_CLOSE (paplift e2)
         in  return CL_NEW (let x = e'1 in e'2)
    else if e == switch swe of {case C1 : e1 ... Cn : en} default edef)
         let swe' = CL_CLOSE (paplift swe)
             e'i = forall i ∈ {1...n} (CL_CLOSE (paplift ei))
             e'def = CL_CLOSE (paplift edef)
         in  return CL_NEW (switch swe' of {case C1 : e'1 ... Cn : e'n}
                                default edef)
    else if e == (if e then e1 else e2)
         let (_, e') = CL_CLOSE (paplift e)
             (_, e'1) = CL_CLOSE (paplift e1)
             (_, e'2) = CL_CLOSE (paplift e2)
         in  return CL_NEW (if e' then e'1 else e'2)
    else if e == Closure n ce {...} pend total
         let ce' = CL_CLOSE (paplift ce)
         in  CL_NEW (Closure n ce' {...} pend total)
    else if e = New n {x1 = e1 ... xn = en)}
         let e'i = forall i ∈ {1...n} CL_CLOSE (paplift ei)
         in  return (New n e'i)
    else return e
```

Listing 6.1: Partial application lift

`f2` is free in `f1`, a lambda lift will generate the closure,

```
Closure "f1" {f2 x} {f2} 0 0
```

for `f1`. The pap lift will generate the Closure

```
Closure "f1" (f2) {x} 1 2
```

for the partial application `f2 x`, which is combined with the first closure to give the result

```
Closure "f2" {f2} {x, f2} 1 2
```

where `f2` is free and `x` is applied.

If the pattern `Closure "_" {Closure "_" ...} ...` is met, we combine the two by preserving the encapsulated expression stored for the inner `Closure`. We want to preserve the free environment in the outer closure because the inner closure will reference the same free environment. We should never encounter a situation where the outer closure is caching a partial application. If we do, then the transformation has been executed incorrectly.

The pap lift "lifts" the computations of the following form:

```
f : Integer -> Integer -> Integer -> Integer;
f = x -> let retf = x -> y -> ...; in retf;
g = x -> ...

f2 = f (g 2) 2
```

In this example, the function `f` is of single arity and returns a function of arity 2. The application `f (g 2) 2` will produce a partial application when we attempt to apply the function return after the initial application `f (g 2)` to the remaining argument. Pap lifting will result in the following code:

```
f2 = Closure "f2" (f (g 2)) {2} 1 2
```

We encapsulate the second application to `2` by creating a Closure that expects the second argument, whatever that happens to be, before applying both the arguments. It is important to note here how we encapsulate the application `f (g 2)`. This was covered in our discussion in section 4.2.2 and is central to the idea of "eager" `encapsulation`.

### *6.1.3 Lambda Doping*

**Definition 6.3.** *The algorithm* `lambdadope` *pattern matches on the term e, with the o-type τ, given in Listing 6.2. It returns the doped expression and the o-type of the expression.*

*We decompose terms such as* `switch` *and* `if` *using* `maybeMangleExpr`*. The dope is decided by comparing the expression's o-type against the supplied o-type. We dope an expression if it is an application, lambda or new expression.*

*We use* `type` *to return the type annotation for some expression e. We access the head and tail of a type by using* $\tau_{head}$ *and* $\tau_{tail}$*, respectively. We use the function* `expandotype` *to unwind the o-type. We sometimes use* $\circ$ *for function composition, where it is not apparent by context. The function* `oarity` *returns the operational arity of the function type, as opposed to its ML arity, which is returned by* `arity`*. We use @ for the application operator and* `isFunction` *returns a boolean if the type is constructed with rightarrow.*

`newName` *generates a list of fresh variable names disjoint from the set of free variables in e.*

We discussed the doping of if and switch expressions in section 5.3.3. The if and switch expressions that appear as the principle function symbol are doped in lines 7-10 and 15-18. We dope the branches against the *o*-type of the if or switch expression by using the call

`maybeMangleExpr (`$e_1'$`, type` $e_1'$`)`

which will "mangle" each branch if there is an impedance mismatch between the *o*-type of the branch and the supplied *o*-type.

We dope applied arguments in lines 19-22. We use the type returned from lambda doping $e_1$ and access the head using $\tau_{head}$. This gives us the dope target *o*-type against which the applied expression's *o*-type is tested, using the call:

$e_2''$ `= maybeMangleExpr (`$e_2$`,` $\tau_{head}$`)`

*Lambda doping example*

We give a simple example of the algorithm. Consider the following:

```
f  = x -> let y -> y + x in y
f2 = foldl f 2 [1, 2, 3];
```

The *o*-type inferred for `f` and the instance of `foldl` in the expression above are given, respectively, as:

```
lambdadope (e : τ)
 = if   e == x
       return (τ, x)
   else if e == (λx.e1)
       let (_, e1') = lambdadope (e1)
           (e1'') = if (e1' == if e then e1 else e2) or
                       (e1' == switch swe of {case C1 : e1 ... case Cn : en}
                                             default edef)
                       then maybeMangleExpr (e1', type e1')
                       else e1'
       in  return (τ, (λx.e1''))
   else if e == e1 e2
       let (τ, e1') = lambdadope (e1)
           (_, e2') = lambdadope (e2)
           (e1'') = if (e1' == if e then e1 else e2) or
                       (e1' == switch swe of {case C1 : e1 ... case Cn : en}
                                             default edef)
                       then maybeMangleExpr (e1', type e1')
                       else e1'
       in if (isFunction τhead)
          then let e2'' = maybeMangleExpr (e2', τhead)
               in  return (τtail, e1'' e2'')
          else return (τtail, e1' e2')
   else if e == let x = e1 in e2
       let (_, e1') = lambdadope (e1)
           (_, e2') = lambdadope (e2)
       in  return (τ, let x = e1' in e2')
   else if e == (switch swe of {case C1 : e1 ... case Cn : en} default edef)
       then let (_, swe') = lambdadope (swe)
                (_, ei'') = forall i ∈ {1...n} lambdadope (ei)
                (_, e'def) = lambdadope (edef)
            in  return (τ, switch swe' of {case C1 : e1' ... case Cn : en'}
                                          default e'def)
   else if e == (if e then e1 else e2)
       then let (_, e') = lambdadope (e)
                (_, e1') = lambdadope (e1)
                (_, e2') = lambdadope (e2)
            in  return (τ , if e' then e1' else e2')
   else if e == Closure n ce {...} pend total
       then let (_, ce') = lambdadope (ce)
            in  return (τ, Closure n ce' {...} pend total)
   else if e = New n {x1 : τ1 = (e1 : τ1') ... xn : τn = (en : τn')}
       let ei'
             = forall i ∈ {1...n}
                 (let (_, ei') = lambdadope (ei)
                  in  maybeMangleExpr (ei, τi))
       in  return (τ, New n ei')
   else return e
```

Listing 6.2: Lambda dope

```
maybeMangleExpr (e : τ, φ)
 = if   e == (if e then e1 else e2)
        let e1' = maybeMangleExpr (e1, φ)
            e2' = maybeMangleExpr (e2, φ)
        in  return (if e then e1' else e2')
   else if e == (switch swe of {case C1 : e1 ... case Cn : en} default edef)
        then let ēi' = forall i ∈ {1...n} maybeMangleExpr (ei, φ)
                 edef' = maybeMangleExpr (edef, φ)
             in  return (switch swe of {case C1 : e1' ... case Cn : en'}
                                default edef')
   else let φ' = expandotype (φ)
            τ' = expandotype (τ)
        in  if (isFunction τ') and (oarity φ' <> oarity τ')
            then let e' = lambdawrap (e : τ, φ')
                 in  return (paplift e')
            else return (e)

expandotype (τ)
 = foldr1 (→) φ
```

Listing 6.3: Maybe mangle expression

```
lambawrap (e : τ, φ)
 = let n = arity φ
       x̄i = forall i ∈ {1...n} newName e
       e' = paplift (foldl1 @ e x̄i)
   in  return (λx̄i.e' : φ)
```

Listing 6.4: Lambda wrap

$Integer \rightarrow \{Integer, Integer\}$
$\{Integer, Integer, Integer\} \rightarrow Integer \rightarrow [Integer] \rightarrow Integer$

After performing a depth first traversal of the expression, we stop the trace at the application for the expression `foldl f`. The *o*-type returned by the call

`(`$\tau$`,` $e_1'$`) = lambdadope (`$e_1$`)`

is

$\{Integer, Integer, Integer\} \rightarrow Integer \rightarrow [Integer] \rightarrow Integer$

The type $\tau_{head}$ is given as

$\{Integer, Integer, Integer\}$

and $\tau_{tail}$ as

$Integer \rightarrow [Integer] \rightarrow Integer$

We then call `maybeMangleExpr`, supplying the right hand expression `f`, the result of which is assigned to the variable $e_2$. `maybeMangleExpr` works by pattern matching on the form of the expression $e_2$. In this case, it is a variable, and we drop through to the last else, and expand the *o*-type of $e_2$ and the value of $\tau_{head}$. This call to `expandotype` is required so we have a type constructed with $\rightarrow$. It makes sure that both *o*-types are consistent, in the sense, that neither one is overrolled. We perform an o-arity test using `oarity`, which, in this case, returns true for the condition resulting in a lambda wrap using the call `lambdawrap`. This creates the expression

```
(a -> b -> f a b)
```

which becomes the new value of the right hand expression. We check to make sure this does not create a new partial application by calling the function `lambdawrap`. The process repeats until we have "consumed" the *o*-type given for the expression. The result will be:

```
f  = x -> let y -> y + x; in y
f2 = foldl (a -> b -> f a b) [1, 2, 3];
```

### *Reducing Dopes in Push/Enter*

There is a further refinement we can make to the doping algorithm given earlier. Recall in Listing 5.7 that if we were reducing via push/enter, we did not need to dope the function `func4`, because the ML arity of its type and the type expected

by `foldr` were the same. We can amend the arity test if we are compiling to push/enter to the following:

```
else let φ' = expandotype (φ)
         τ' = expandotype (τ)
     in  if (isFunction τ') and
            (arity φ' <> arity τ')
         then let e' = lambdawrap (e:τ, φ')
              in  return (paplift e')
         else return (e)
```

Listing 6.5: Ammended arity test for push/enter

The test for whether we require a dope is now a comparison between the ML arities. It is interesting to note that this will produce less dopes than testing the operational arity. We give a reason for this in section 7.4.2.

## *6.2 A Trivial Optimisation*

Consider this situation:

```
func3 : Integer -> (Integer -> Integer) -> Integer;

func1 = let //f1 : Integer -> Integer -> Integer;
            f1 = <something>;
            f2 = map f1 [1..n];
            f3 = map f1 [1..n];
        in  ...f2...f3...
```

Note we use `...` to denote that the context is not important in the example. We generate the same dope for each occurrence of the function `f1` in the body of the function `func1`. We can improve this, and instead generate a single `let` bound dope and pass that to every callee. This works because the lambda will generate a new Closure when it is applied inside the body of the callee. The example is rewritten as:

```
func1 = let f1_dope = (a -> Closure (f1) {a} 1 2);
            f1 = <something>;
            f2 = map f1_dope [1..n];
            f3 = map f1_dope [1..n];
        in  ...f2...f3...;
```

This is an obvious optimisation that prevents code duplication. It also allows us to *float* the dope, which we discuss in more detail in section 6.4.1.

## 6.3 Summary

The problem with doping is the reduction in efficiency introduced by the creation of extra lambdas. The number of extra lambdas is equal to the number of type impedance mismatches. Further analyses are reserved for future work.

We have shown that the number of dopes using the reduction semantics for push/enter are less than those generated for eval/apply. We have also shown by case analysis that we can produce a operational coercion that maintains the correct reduction semantics for every unknown application.

In section 6.4, we discuss the interaction of lambda doping with other transformations carried out by a typical optimising compiler such as GHC. In the case of a lazy language in which we can prove, via strictness analysis, that the thunk is sure to be evaluated, we can separate the complete application component from the partial application, using the let-to-case transformation. Lambda doping does not adversely affect other standard transformations, including let-floating, worker/wrapper and case elimination.

Lambda doping is trivially confluent and terminating. Confluence is easily verified by the fact that we only produce extra lambdas which saturate function calls. The algorithms `lambdadope` and `paplift` terminate because we only apply them at most once to an expression.

We now have a method of saturating all applications. This now makes compiling to the system stack via $C^\sharp$ on the CLI a viable option.

## 6.4 Optimisations

Optimisations are applied in sequence by successively applying a set of transformations. Compilation by transformation is a well-researched field (Jones & Marlow 2002), (Jones et al. 1997), (Wansborough & Peyton Jones 2000), (Jones et al. 1996), (Marlow 1993), (Appel 1994), (Serrano 1997), (Danvy & Schultz 1997). In most transformation phases, several invariants are maintained to prevent transformations from changing the operational semantics or the meaning of the program. Transformations suggest a notion of refinement and equivalence that allow the compiler to make valid transformations that maintain the semantic equivalence of the terms. As such it is instructive to briefly consider how any additional transformations may affect those already available in the GHC compiler, such as:

- performing a number of generic transformations, including inlining, collectively implemented in the *simplifier*,
- performing full laziness, let-floating and local floating, reapply the simplifier,
- performing strictness analysis and CPR analysis, reapply the simplifier,
- performing a worker/wrpper transformation, based on the information from both these analysis,
- reapplying the simplifier, which inlines the wrappers at each call site and using the extra detail at each call site to further simplify calls to our desired form.

Some optimisations are affected by the extra presence of lambda boundaries that doping injects. Note that in this section, we use Haskell syntax to ease the notational burden. It is important to remember that these transformations are also applicable in Mondrian. Lambda doping is intuitively a "once" only pass, unlike, for example, those included in GHC's simplifier, which are record label elimination, shrinking inlining, dead variable elimination and beta reduction (Appel & Trevor 1997), (Serrano 1995). "Discovering" more lambda doping transformation reduxes after applying an additional optimisation pass would imply a breakdown in the subject reduction property, and would therefore be incorrect. GHC includes a check to make sure optimisations do not break subject reduction by checking the correctness of the core type calculus. In GHC, the unifying optimisation framework theory is that it progresses to the compilation target by transformation (Jones 1996). We take a quick look at full laziness and strictness analysis and their interaction with doping.

### *6.4.1 Full Laziness*

Full laziness reduces the overhead associated with extra allocation of THUNKS by doing the reverse of the floating inwards operation. Consider the following example:

```
f = \xs -> let g = \y -> let n = length xs
                         in ....g....n....
           in  ...g...
```

We use the syntax `...g...` to indicate that the immediate context of the expression is not important to the application of the transformation. `length xs` is

recalculated on every recursive call to `g`. If we move the binding for `length xs` outside of the `let` binding then `n` is not recalculated every time we call `g`, resulting in the following:

```
f = \xs -> let n = length xs
           in  letrec g = \y -> ...g...n...
               in      ...g...
```

This is different from lambda lifting. Jones et al. in (Jones et al. 1996) shows how to decouple full laziness from lambda lifting by regarding it as an exercise in floating bindings outwards.

The float-in transformation seeks to avoid pushing let bindings inside lambdas. In contrast, the full laziness transformation actively wants to push the let binding outside the lambda. Lambda doping gives us the opportunity to realise the "full potential" of full laziness by separating total compilations from partial applications.

Traditionally, application of the full laziness transform is controlled by the following, given in (Jones et al. 1996):

1. avoid floating out bindings that are already values or can be reduced to values with negligible effort, because this reverses the benefits of the let-in floating.

2. using the lambda abstraction no more then once. It makes no sense to float outside a lambda which is used only once.

Consider the following doping, where the function `f1` is annotated with its structural type:

```
// f1 : Integer -> Integer -> {Integer, Integer, Integer};

f = \x -> \y -> let g = \x -> let h = map (f1 2 3 4) [1...n];
                              in  h 2 x;
                in  ...g...g...
```

We will dope the expression `f1 2 3 4`, giving us:

```
(\a -> Closure (f1 2 3) {4, a} 0 2)
```

Using the identity $let\ x = a\ in\ e \equiv\ e[x/a]$, we can view the Closure as:

```
Closure (let x = f 2 3 in x) {4, a} 0 2
```

Applying the full lazy transform moves the let bound expression out of the Closure, giving us:

```
f = \x -> \y ->
      let n = f1 2 3
      in  let g = \x ->
                    let h = map (\a -> Closure (n) {4, a} 1 2)
                                  [1...n]
                    in  map h [1...n];
          in  ...g...g...
```

Listing 6.6: Lazy dope

We now save ourselves the reallocation of a THUNK every time we call `g` in the body of the `h`. This transformation applies to any application in which we can separate it into a total and partial computation. It applies despite the occurance of an impedance match (as is true in the example above) and where we can prove that the computation is used more then once, for example, when using usage analysis. If the computation we are floating out involves a call to another recursive binding, then we lift into the let.

We mentioned earlier that GHC includes a simplification phase which includes an inlining phase. The decision to inline an expression is based on the idea of the *occurrence* or *usage count*. Given the expression `let x = E in B`, the occurrence of `x` is the number of times it occurs in `B`. Usually, an inline is deemed "safe", if it occurs once and does not occur within a lambda form. We can inline into a lambda form if the inlined expression is *bounded* in some manner. An expression is usually considered to be bounded, if it is a variable, constructor application, lambda abstraction or an expression that is sure to diverge. In Listing 6.6, inlining will not move the `let` binding back into the Closure because to do so requires moving through the lambda bound to `g`, which will duplicate work. We have to take care not to inline work that we have extracted from a dope. Consider the following:

```
f = \x -> \y -> let g = f1 2 3 4;
                in  ...g...g...
```

This is translated to:

```
f = \x -> \y -> let n = f 2 3;
                    g = (Closure (n) {4} 1 2)
                in  ...g...g...
```

Notice that the `n` is not inlined within the Closure because it is a total application.

### *6.4.2 Strict Dopes*

The segregation of total and partial applications allows us to more effectively exploit strictness analysis. There is a wealth of information on strictness analysis including (Wadler & Hughes 1987), (Wadler 1988) and (Nocker 1993). The current GHC implementation uses a strictness analyzer based on *backwards* demand analysis. This is discussed in (Jones et al. 2004) and uses an abstract interpretation which records "demands" that the function places on its arguments, its environment and on its result. Take, for example, the following:

```
f : [a] -> [b] -> Bool
f = \xs -> \ys -> null xs && null ys;
```

The function is annotated with the demand signature `SL` on its arguments `xs` and `ys`, respectively. Where `S` denotes that `xs` is strict in `f`, and `L` denotes `ys` is lazy in `f`, we cannot guarantee that `ys` will be evaluated.

Jones et al. take this transformation a step further by defining the *call demand*. Consider the curried application `(f x y)`, which really means `((f x) y)`. If demand `d` is placed on this expression, which demand is placed on the sub expression `(f x)`? GHC assigns the call demand `S(d)`, therefore the demand on the full expression is `S(S(d))`. The full abstract interpretation used in demand analysis is outside the scope of this discussion, so we assume that strictness analysis has annotated the expressions appropriately. GHC exploits strictness information in a number of ways. Consider the following:

```
f x y = if x then p else q
        where (p, q) = h y
```

This is transformed naively to:

```
f x y = let t = h y
        q = case t of (p,q) -> q
        p = case t of (p,q) -> p
    in  if x then p else q
```

The strictness analysis will discover that `q` and `p` are sure to be evaluated, and, as such, will perform a *let-to-case* transformation, giving us:

```
f x y = case (h y) of
         (p1, q1) -> let p = case p1 of (p, q) -> p
```

```
                        q = case q1 of (p, q) -> q
                    in  if x then p else q
```

Subsequent cancellation will eliminate the inner cases, giving us:

```
f x y = case (h y) of
          (p1, q1) -> if x then p1 else q1;
```

GHC also uses a variation of the let-to-case transformation called the worker/wrapper transform discussed in (Jones 1996). This transformation separates the computation performed by the function from the reduction by reducing the arguments within a wrapper function. The wrapper function then calls the worker, passing it reduced arguments. The obvious duplication of work is avoided by aggressively inlining the wrapper into the caller sites. We can make use of the let-to-case transformation in the following:

```
f = let f2 x = let f3 x = ...
                in  (\x -> y -> f3 x + y)
    in  map (f2 2 3) [1, 2, 3]
```

In *this* case, we know `map` is strict in its first argument. We can therefore rewrite the encapsulated expression as:

```
f = let f2 x = ...
    in  map (case ((f 2) 3) of x ->
                (\a -> \b -> Closure (x) {} 2 2))
                    [1, 2, 3]
```

The `case` expression reduces the expression `((f 2) 3)`, and binds the result to the variable `x`, which is stored in the Closure. The example given earlier in Listing 6.6 will be transformed to:

```
f = \x -> \y ->
      case (f1 2 3)
      {  n -> let
                g = \x ->
                      let h = map (\a -> Closure (n) {4} 1 2)
                                  [1...n]
                      in  map h [1...n];
              in ...g...g...
      }
```

# Chapter VII

# Front-End : The Mondrian Type System

In this chapter, we will discuss the type system in the initial version of Mondrian. We will then extend the grammar of the language to support type annotations and parametric polymorphism. From there, we can define a type system to infer implicit widening coercions and *o*-types.

## *7.1 Dynamically Typed Mondrian*

Mondrian was initially "dynamically" typed, therefore all type errors were only detected at runtime. We noted that we can treat all values as of type *Object*. The CLI will perform runtime checking of the type tag when the value is used in an operation. This is a very simple approach and works adequately for scripting.

Static type information, however, is useful and worth the effort to infer. Type information, as we remarked earlier, affords us a level of safety because well-typed programs do not go wrong. The lack of type information also restricts our ability to perform optimisations and check that they are correct (Boquist 1999), (Jones 1996), (Baker-Finch et al. 2004) and (Gill et al. 1993).

## *7.2 Mondrian with Static Types*

We extend the Mondrian language, given in Definition 3.1, to accommodate user type annotations and parameterised data types. The grammar for the new syntactic forms are given in Definition 7.1. Mondrian uses classes with generic parameters to represent polymorphic sums. We can say that Mondrian is a generic *producer*, because it can define generic classes. It can also consume generics by instantiating parameterised classes with their concrete arguments.

### *7.2.1 Type Annotations*

We can annotate Mondrian terms using the following syntax. We have already encountered type annotations in earlier examples, but we formalise the notation here:

```
map : forall a, b => (a -> b) -> [a] -> [b];
map f xs
 = switch xs of
   {    case Nil : [];
   ;    case x::xs : (f x) :: (map f xs);
   };
```

The type `forall a, b => ...` explicitly quantifies the type variables `a` and `b`. An explicit type is useful beacuse it gives us a "sanity" check to test against the inferred type. If the two match, then we have a successful typing, otherwise, we return an error. The `forall` syntax indicates that we are quantifying the type variables `a` and `b`.

### *7.2.2 Sum Types*

We discussed how type constructors were used in Mondrian in section 2.1.3. Recall here the Haskell definition for the BinaryTree:

```
data Maybe v = Nothing | Just v

data FMBT k v
    = E
    | BT (FMBT k v) k (Maybe v) (FMBT k v)
        deriving (Show, Read, Eq)
```

Listing 7.1: BinaryTree, defined in Haskell

With parametric datatypes, we are now in a position to define the following:

```
class Maybe<V> {};
class Nothing : Maybe<V> {};
class Just : Maybe<V> { V : value; };

// BinaryTree definition
class FMBT <K, V> {};
class E <K, V> : FMBT <K, V> {};
class BT <K, V> : FMBT {    FMBT<K,V> left;
                            K key;
```

```
        Maybe<V> v;
        FMBT<K,V> right;
};
```

Listing 7.2: BinaryTree, defined in Mondrian with generics

The type `FMBT` is parameterised by the type variables `K` and `V`. The `Maybe` data type is parameterised by `V`. We note that the type of the variable `value` is now `V` instead of `Object`. We could now instantiate instances of `Maybe` using the following Mondrian sample:

```
f = let x = Just<Int>{ ... };
    in  ...
```

Listing 7.3: New Just instance

Alternatively, it should be possible to infer the type of the constructor's parameters by using the type of the expression assigned to the variable `value`. This would make the line `Just {...}` adequate and would effectively instantiate the data type to `Just <Int> {...}` as if we had done so explicitly.

We discussed in section 2.1.3 how we used the `switch` statement to project fields from sum values. If we are switching on a reference type, the type of the scrutinee should be a subtype of the least upper bound (lub) of the case types. For a value type, the scrutinee type is the same as the case types.

### *7.2.3 Product Types*

We discussed product types in section 2.1.4. With parametric data types, we are also in a position to define parametric product types. Given the expression `let a = (1, 2) in ...`, Mondrian will construct a value of type `Pair`, bound to `a`. Pair can be defined as:

```
class Pair <A, B> {A a, B b};
```

In the expression given above, it would be instantiated in the following way:

```
Pair<Integer, Integer> a = new Pair<Integer, Integer>(1, 2);
```

### *7.2.4 Coercions*

Mondrian currently checks widening coercions dynamically, but we want the type checker to be able to infer these statically. In SML, coercions are only applied implicitly when invoking .NET methods. Otherwise, we must explicitly provide a

cast using the syntax `exp :> ty`, where exp is the object to cast to a superclass type ty. Downcasts are provided using the same syntax, which may result in a InvalidCastException.

Unlike $F^\sharp$ and SML.NET, Mondrian infers implicit coercions between .NET types and Mondrian types. There is no need for an explicit coercion when one can be inferred by the type system. However, Mondrian, at present, lacks an explicit cast syntax for downcasting types of the form of SML.NET, given as `exp :> ty`, and in $F^\sharp$ as `(cast exp : type)`. Down casts can be achieved by using the `switch` syntax and testing the type of the value directly. $F^\sharp$ also does not recognize simple subtyping constraints, and up casts are explicitly required.

For example, in Mondrian:

```
nodetoxmldata :: XmlNode -> Maybe
nodetoxmldata n
 = switch n of
   {    XmlElement { ... } :

   ;    XmlCharacterData { ... } :
   }
```

Mondrian tests statically if the types `XmlElement` and `XmlCharacterData` can be unified, that is, if they share a common least upper bound, in this case, `XmlNode`. If the runtime type of `n` is either `XmlElement` or `XmlCharacterData`, then the appropriate alternative is chosen. Otherwise, we can choose to throw an appropriate exception.

An interesting note is that SML does not allow $C^\sharp$'s numeric widening coercions to be implicit, that is, it does not allow the natural subsumption between types unless it is provided using an explicit coercion. In Mondrian, this coercion is implicit, as the Integer type can be regarded as Int64 type, as long as the constraint is made explicit in the environment.

### *7.3 Mondrian Type Language*

Based on our language requirements, we state the following type system requirements:

1. Type language, supporting first order parametric quantification, data declarations, type annotations, function types, and *o*-types

2. Subsumption for implicit widening coercions over reference types, where valid

3. An inference algorithm, reconstructing $o$-types from $\mathrm{M}_{CM}$ terms, and accepting user type annotations.

### *7.3.1 Notation*

The syntax of the Mondrian type language is given below in Definition 7.1.

**Definition 7.1.** *The type language for type checking Mondrian terms with operational typings and primitive subtyping.*

$$
\begin{array}{rrcll}
\textit{types} & \tau, \phi & ::= & \textit{class } T\ \overline{\alpha_j} \ = \ \overline{C_i\ \overline{\tau_{ik}}} & \textit{sum} \\
 & & & \iota_1 \mid \ldots \mid \iota_n & \textit{ground} \\
 & & \mid & \sigma_1 \rightarrow \sigma_2 & \textit{mapprod} \\
 & & \mid & \alpha, \beta & \textit{variable} \\
 & & \mid & (\sigma_1 \ldots \sigma_2) & \textit{prod} \\
 & & \mid & \textit{error } n & \textit{error} \\
 & & & & \\
\textit{types} & \kappa, \rho, \nu & ::= & \{\tau\} & \\
(\textit{operational}) & & & & \\
 & & & & \\
\textit{scheme} & \sigma & ::= & \textit{forall}\overrightarrow{\alpha}.\tau & \textit{scheme} \\
(\textit{schemes}) & & & &
\end{array}
$$

We use $\alpha$ and $\beta$ to stand for type variables. We use $e$, $t$ and $f$ to range over terms and $x$ and $y$ to range over term variables. $o$-types are ranged over by $\kappa$ and $\rho$ and type schemes denoted by $\sigma$. Given the $o$-type $Integer \rightarrow \{Integer, Integer\}$, we reconstruct the equivalent ML type by unrolling the tail $\{\}$ constructor, using the function `expandotype` defined in Definition 6.3. Thus $o$-types and ML types are equivalent up to `expandotype`. We differentiate between the arity of a $o$-type and a ML type, calling the former the o-arity and the latter the ML arity, or, more simply, just the arity. We use $\mathcal{TV}$ to denote the set of type variables and $\mathcal{T}$ the set of all types. We use $FTV(\tau)$ to denote the set of free type variables in $\tau$. A type $\tau$ is "free" if it is not explicitly quantified by a type scheme. The set of ground types $i_1, \ldots, i_n$ are given in section 2.1.2. We denote the set of operational types as $\mathcal{ST}$.

### *7.3.2 Generic Mondrian Grammar*

We can extend the Mondrian grammar, given in Definition 3.1, to include user type annotations and parametric data types.

**Definition 7.1.** *The $M_{CM}$ grammar extended to parameterise data definitions.*

$$(\textit{declarations}) \quad d \quad ::= \quad \textit{class } T\ \overline{\alpha_j}\ T_1\ \overline{v_i : \tau_i} \mid \textit{import } v \mid v\ =\ e \mid v\ :\ \tau$$

The type variables $\overline{\alpha_j}$ can appear in the types $T$, $\overline{\tau_i}$ and $T_1$. We can now declare classes of the form:

```
class A<Integer> : B<Integer> {};
```

### *7.3.3 Type Contexts*

A type context $\Gamma$ is any finite, perhaps empty, set of type assignments to terms $\Gamma = \{x_1 : \alpha_1, \ldots, x_m : \alpha_m\}$ whose subjects are term-variables and which is *consistent* in the sense that no variable is the subject of more than one assignment. Standard rules define well-formed contexts and well-kinded types which are omitted here. We write $\Gamma, x : \sigma$ for the extension of the set $\Gamma$ with the element $(x : \sigma)$. We write $\alpha.x \in \Gamma$ to express that the term variable $x$ is found in the context $\Gamma$ (with some unspecified type), and $\alpha \in \Gamma$ that the type variable $\alpha$ is found in the context $\Gamma$. We write $\Gamma / x : \sigma$ for the context $\Gamma$ without the element $x$.

A type context can be modified by applying a type substitution. A type substitution is any expression $(\langle\alpha_1, \sigma_1\rangle, \ldots, \langle\alpha_n, \sigma_n\rangle)$ where $\alpha_1, \ldots, \alpha_n$ are type variables and $\sigma_1, \ldots \sigma_n$ are any types. A substitution $S$ is any expression $[\sigma_1/\alpha_1, \ldots, \sigma_n/\alpha_n]$, where $\alpha_1, \ldots, \alpha_m$ are distinct type-variables and $\sigma_1, \ldots, \sigma_n$ are any types. For any type $\tau$ define $S(\tau)$ to be the type obtained by simultaneously substituting $\sigma_1$ for $\alpha_1, \ldots, \sigma_n$ for $\alpha_n$ throughout $\tau$. Substitutions are extended over composite types in the obvious way, via structural induction. For any type context define $S(\Gamma)$ as the point-wise application $S(\langle x_1, \sigma_1\rangle, \ldots, \langle x_n, \sigma_n\rangle) = \langle x_1, S\sigma_1\rangle, \ldots, \langle x_n, S\sigma_n\rangle$.

### *7.3.4 Type Schemes*

We allow first order quantification over types, as described in (Damas & Milner 1982) and (Harper et al. 1987). We use *type schemes* to denote this set of possible instantiations.

In brief, the polymorphic type system allows certain variables to be used with a set of different types. For example, if the identity function $\lambda x.x$ is `let`-bound to

the variable `f`, then `f` may be used as a function of type $Integer \rightarrow Integer$ and applied to values of type $Integer$. Alternatively, `f` may be used as a function of type $Bool \rightarrow Bool$, and applied to values of type $Bool$. In a more complicated example, the function `compose`, typed as $(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma$ can be applied to any two functions where the output from the first can be applied to the second. For example, we can apply `compose` to the arguments `+ 2`,`- 2` and `2` which would produce the result 2. In curry style typing, a type can consist of type variables to which an infinite number of possible types can be assigned. Restrictions exist, in which a type variable can be constrained to only be instantiated within certain limits, however, we will not be investigating these.

Generally, any type $\tau$ with $\overrightarrow{\alpha} = FTV(\tau) \neq \varnothing$, can be quantified to the type scheme $\sigma = \forall\overrightarrow{\alpha}.\tau$. The set $FTV(\tau)$ is the set of free type variables in $\tau$, defined in the usual way. $\overrightarrow{\alpha}$ denotes the list of type variables $\alpha_1, \ldots \alpha_n$, and $\forall\alpha_1, \ldots, \alpha_n.\tau$ binds the type variables $\alpha_1, \ldots, \alpha_n$ in $\tau$. We note that the type scheme $\sigma$ can also denote the empty set of type variables, where $\sigma = \tau$. We define the specialisation relation $\triangleright$ over types by considering the type $\tau$ the specialisation of a type scheme $\sigma = \forall\alpha.\tau$ by some substitution $S$, where $S$ is a partial homomorphism of type $TV \rightarrow T$. This is written as $\sigma \triangleright \tau$. The substitution $S(\sigma)$ is capture-avoiding, that is, we do not substitute a free type variable for one that is bound by the type scheme. We refer to the process of applying a substitution $S$, substituting a ground type for each type variable in a type scheme, as *instantiation* and the resulting type the *instantiated* type.

We can extend specialisation in the usual way, that is, $\sigma \triangleright \sigma'$, if whenever $\sigma' \triangleright \tau$ then $\sigma \triangleright \tau$. For example, the type scheme $\forall\alpha.\alpha \rightarrow \alpha$ can be specialised to the type $Integer \rightarrow Integer$. We can further generalize this by defining $\triangleright^+$ as the reflexive, antisymmetric and transitive closure of $\triangleright$.

### *7.3.5 Simple Type Rules for Functional Mondrian*

We are now in a position to define our initial set of type rules for the "functional" aspects of Mondrian. By "functional", we mean the rules for typing variables, abstraction, application and let. These are not syntax-directed and are defined here to give us a set of type rules to which we can investigate the addition of subtyping and *o*-types on the CLI. A set of syntax-directed functional type rules are given in section 7.8.

We list a few definitions before continuing:

**Definition 7.3.1.** *We express a* subtype statement *as a phrase of the form* $\Gamma \vdash \sigma \leq \tau$*, where* $\sigma$ *and* $\tau$ *are types. A* typing statement *is a phrase of the form* $\Gamma \vdash \ e : \sigma$*, where* $e$ *is a term and* $\tau$ *is a type. The* body *of the statement is the portion to the right of the turnstile.*

**Definition 7.3.2.** *A* derivation *of a subtyping or typing statement* $J$ *is a proof tree, which is valid, according to some collection of inference rules whose root is* $J$*. We write* $d :: J$ *to indicate that* $d$ *is a derivation* $J$*.*

**Definition 7.3.1.** *Functional type rules for* $M_{CM}$ *without subtyping.*

$$\frac{}{t : \tau \vdash t : \tau}\, T - VAR$$

$$\frac{x : \alpha, \Gamma \vdash y : \beta}{\Gamma \vdash (\lambda x.y) : \alpha \rightarrow \beta}\, T - ABS$$

$$\frac{\Gamma \vdash f : \alpha \rightarrow \beta \quad \Gamma \vdash x : \alpha}{\Gamma \vdash f\ x : \beta}\, T - APP$$

$$\frac{\Gamma \vdash e' : Close(\sigma, (\Gamma, x : \sigma)) \quad \Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (let\ x = e'\ in\ e) : \tau}\, T - LET$$

$$\begin{aligned} Close\ (\tau, \Gamma) \quad &\equiv\ \forall \alpha_1, \ldots, \alpha_n.\tau \\ &\quad where\ \alpha_1, \ldots, \alpha_n = FTV(\tau)/FTV(\Gamma) \end{aligned}$$

We use the function *Close* to build a type scheme quantifying over those variables that are free in the environment $\Gamma$.

## *7.4 Subtyping*

Our second requirement is that we be able to statically infer widening coercions over types. Subtyping, as we have discussed earlier, is a complex field in practical applications, hence we largely restrict its application in this work.

The $\text{CLR}_{\leq}$ uses a subtype ordering constructed using inheritance over class and interface types. An interface is similar to a class, except it does not carry any state or methods. Given two types, $\alpha$ and $\beta$, and the constraint $\alpha \leq \beta$ established by inheritance, the $\text{CLR}_{\leq}$ treats values of type $\alpha$ as a subset of the values described

by $\beta$, and allows implicit coercions "up" the subtype order and explicit coercions "down" the order, the later checked during runtime. We have already met several subtype orders constructed using inheritance. Consider our definition of `FMBT` in the example given earlier in Listing 2.2. The constructors are derived from `FMBT` by specifying the parent type in the class definition. In addition, the CLI defines a universal top type called *Object*, which is the lub of every type. This effectively gives us a semi or quasi-lattice, that is, one in which every pair of types shares a least upper bound.

This approach is not without precedent and is discussed in (Nordlander 1998) under the term *pragmatic subtyping*. The subtype relation in a pragmatic subtype system, unlike the usual structural subtype relation discussed in (Fuh & Mishra 1988) and (Smith 1994), is defined explicitly between names. As such, a constraint $Integer \leq Real$ holds only if the environment $\Gamma$ or some coercion set $C$ contains the same constraint or one that can be derived by structural decomposition. We call this the environment condition over the subtype relation. With this invariant, the subtype relation is consistent with that defined in the $\text{CLR}_{\leq}$. We can make a further restriction here with respect to interface types. Consider two types which share a common interface, with the lub of the two types being the interface type. Typing an argument as the interface type means we can take arguments of both types. While this is valid, we restrict the lub to only class types. This is done purely to simplify coercion inference and could be extended in any future work.

Traditional subtype type systems allow structural decomposition over all composite types, including the $\rightarrow$ operator, (Smith 1994) and (Pottier 2001). However, this assumes a subtype relation which observes the usual contra/covariance of the argument and return types. The $\text{CLR}_{\leq}$ restricts the subtype relation so that it is covariant over all types.

We can decompose quantified sum types in the usual way. Given the constraint, $Cons\langle B\rangle \leq List\langle A\rangle$, we can satisfy this only if the constraints $B \leq A$ and $Cons \leq List$ are satisfied. In a constrained subtype system, (Smith 1994) this structural decomposition is usually computed from the constraint graph using a *closure* algorithm. Finding the closure of a constraint set is a process whereby constraints over composite types are broken down into a set containing simple atomic constraints. However, we do not build a constraint graph, but simply check the validity of the coercion during inference.

Our restrictions cause types inferred usually to be less general than those inferred using a constraint based system. Consider the following example:

```
g = f -> x -> (f (f x))
```

The constrained type would be given as:

$\forall\alpha\beta.(\alpha \to \beta) \to \alpha \to \beta \mid \beta \leq \alpha$

However, the more usual ML type would be inferred in our system as:

$\forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$

We can see that the first type is more general. However, as we have already discussed, practical constraint-based typing algorithms are complex and we did not want complicate inference of *o*-types by posing the problem in a practical constraint-based type system.

The addition of a subsumption rule gives us the usual "threeplace relation". This means that every subtype statement will now have the form $\Gamma \vdash \phi_1 \leq \phi_2$. This is equivalent to $\phi_1$ is a subtype of $\phi_2$ under assumptions $\Gamma$ if neither types are constructed using $\to$.

As a consequence of the environment condition over subtypes, the subtype relation in Mondrian is a partial order, with the usual properties, namely, reflexitivity, antisymmetry and transitivity. It is a partial order because we have no equivalent permutation rule for records (Cardelli & Mitchell 1994) due to name equivalence. We formalize the relation in Definition 7.4.1.

**Definition 7.4.1.** *Subtyping rules for $M_{CM}$.*

$$\frac{\Gamma \vdash \alpha \leq \ Object}{} \ SA - TOP$$

$$\frac{\Gamma, \alpha \leq \beta \vdash \alpha \leq \beta}{} \ SA - NAME$$

$$\frac{class\ T\ \overline{\alpha}\ =\ \overline{C_i\ \overline{\tau_{ij}}} \qquad \alpha_k \in\ FV(\tau_{ij}) \Rightarrow \tau_k \leq \tau_k^{'}\ forall\ k\ i\ j}{T\ \overline{\tau_k} \leq T\ \overline{\tau_k'}} \ SA - CON$$

We add the rule T-SUB to our set of non-syntax-directed type rules for functional Mondrian, defined as:

$$\frac{\Gamma \vdash t : \tau \qquad \Gamma \vdash t : \tau \leq \phi}{\Gamma \vdash t : \phi} \ T - SUB$$

The obvious omission is the lack of a subtype rule over function types. Neither $\text{CLR}_{\leq}$ nor $\text{CLR}_{\leq,\forall}$ provide an explicit way of statically expressing subtyping directly over function types. This prompts the question of whether or not we can efficiently mimic subsumption over function types without an explicit function subtype rule. Consider the simple artificial example with the constraint $\phi \leq \tau \in \Gamma$:

```
g : (τ -> φ);
f : (φ -> τ) -> φ -> τ;
f f2 y = f2 y
```

In a system with subtyping over function types, the function `f2` should be able to accept a function of type $(\tau \to \phi)$. This is due to contravariance in the argument type and to covariance in the return type. This decomposition is expressed in the rule SA-ARROW.

$$\frac{\Gamma \vdash \alpha_1 \leq \beta_1 \qquad \Gamma \vdash \beta_2 \leq \alpha_2}{\Gamma \vdash \beta_1 \to \beta_2 \leq \alpha_1 \to \alpha_2}\, SA - ARROW$$

The antecedents of the SA-ARROW rule place conditions on the argument and return types. Instead of relying on an application of the rule SA-ARROW at the end of a derivation, we can change the term so that antecedents in the SA-ARROW rule occur further up in the type derivation.

Consider the following type deviation for the code `f g l`. We use the simple type rules extended with the subsumption rule SA-ARROW. We use the rule SA-ARROW to promote the type of `g` to that expected by `f`, given in Figure 7.1. We can modify this derivation by deriving it over the expression:

```
f (x -> g x) l
```

The type derivation is given in Figure 7.2. We have elided the derivation of

$$\dfrac{\dfrac{\Gamma \vdash f : (\phi \to \tau) \to \phi \to \tau \qquad \dfrac{\Gamma \vdash g : \tau \to \phi \qquad \tau \to \phi \leq \phi \to \tau \in \Gamma}{\Gamma \vdash g : \phi \to \tau}\, SA - FUNC}{\Gamma \vdash f\ g : \phi \to \tau}\, T - APP \qquad \Gamma \vdash l : \phi}{\Gamma \vdash f\ g\ l : \tau}\, T - APP$$

Figure 7.1: Type derivation for expression `f g l`

$$\dfrac{\dfrac{\Gamma \vdash f : (\phi \to \tau) \to \phi \to \tau \qquad \dfrac{\Gamma \vdash (g\ x) : \tau \qquad \Gamma \vdash x : \phi}{\Gamma \vdash (\lambda x.g\ x) : \phi \to \tau}\, T-ABS}{\Gamma \vdash (f(\lambda x.g\ x)) : \phi \to \tau}\, T-APP \qquad \Gamma \vdash l : \phi}{\Gamma \vdash (f\ (\lambda x : g\ x)\ l) : \tau}\, T-APP$$

Figure 7.2: Type derivation for expression `f (x -> g x) l`

$g\ x$ for clarity, which can be built by an application of $g$ to $x$ using $T-APP$, giving us the expression $(g\ x) : \phi$. We can then apply $SA-NAME$ to promote its type to $(g\ x) : \tau$. Notice that we are structuring the type coercions over the new lambda. While this gives us a feasible way of passing functions, it does not scale efficiently. Consider a more complicated artificial example:

```
g  : τ -> φ;
h  : ((φ -> τ) -> τ -> φ);
f2 : ((τ -> φ) -> φ -> τ) -> (τ -> φ) -> φ -> τ;
f2 x y z = x y z;
```

If we were to derive a coercion without using SA-ARROW, then an expression of the form `f2 h g l` would be rewritten as:

```
f2 (x -> y -> h (t -> x t) y) g x
```

We generated the outer coercion from the elimination of SA-ARROW to the application of `h` to `f2`. The inner coercion `(t -> x t)` is generated because `h` will be passed a function of type $\tau \to \phi$ when it expects a function of type $\phi \to \tau$. The elimination of SA-ARROW generates the inner coercion.

We can see the complexity has increased linearly for every elimination of the rule SA-ARROW. In this case, we had applied it twice and have produced two new lambda coercions. We are already increasing the number of additional lambdas by using lambda doping. We may have to generate more for every application of SA-ARROW, which is computationally expensive. It would also require any clients to generate the same wrappers, but this is less of a problem since the requirement is not solely a Mondrian restriction.

We will set aside this attempt for the moment, and have a look at a different approach, first observing that we have a subtype semi-lattice, and therefore, we can coerce any type to *Object*. If we erased the type signature of `f` to $Object \to \phi \to \tau$, then the call `f g` would proceed normally, that is, without the generation

of a lambda wrap. The problem with this approach, however, is that we require an explicit downcast when compiling to the CLR to use the function `p` at the application `p y`. This is not trivial, as the type of `y` can be any subtype of the type statically inferred for `f`'s argument type. The only way to determine the type of `p` is to query for it at runtime using the reflection libraries. This is operationally expensive, as we would require a runtime query every time we wanted to use a function passed as a first class value.

Hence, the only feasible solution is to use structural coercions by introducing lambda wrapping. However, as we noted, this incurs large operational costs. With covariant atomic coercions, we already have enough power to support the static typing of the Mondrian language. Additionally, with the recent advent of $\text{CLR}_{\leqslant,\forall}$, there is now support for an explicit contra/covariant rule over function types. Therefore, we elect not to support proper contra/covariant subsumption over function types in the present version. It is left for future work.

### *7.4.1 Operational Types*

We have discussed how *o*-types are used and their formal notation. Now, we specify how they are generated. *o*-types are generated by the algorithm given in Definition 7.3. Their generation is divided into two parts: the annotation of the return type in the typing rule TA-ABS, which is defined later, and the function `otype`. We shall delay a detailed example of their generation until we have presented the inference system.

### *7.4.2 Operational Coercions*

In this section, we formalize the notion of an *operational coercion*, which we introduced in section 5.2. We motivate this discussion by recalling our example given for conditional doping in Listing 5.7, which we reproduce below:

```
foldr (if (boolFlag2) func3 else func4) 2 x1;
```

In our example, the line above is doped to the following:

```
foldr (if (boolFlag2)
       then func3
       else (a -> b -> func4 a b) 2 x1)
```

We noted that if we are compiling to a reduction environment in which the responsibility of runtime argument checking was placed on the callee, as in push/enter, then we would not need to dope the function `func4` when passing it to `foldr`.

A dope, or operational coercion, is produced in eval/apply when the *o-type* is not equivalent to the type of the function specified by `foldr`'s first argument. A dope is produced in push/enter when the arity of the *ML type* is not the same as that specified by `foldr`.

We distinguish formally between this stronger requirement on doping in eval/apply and then on push/enter. Intuitively, *o*-types contain more operational information than ML types. This is similar to the use of a 2-pt subtype domain used by Wansbrough & Peyton Jones in (Wansbrough & Peyton Jones 1999) to express usage analysis. More specifically, they use the domain containing 1 and $\omega$ with the order $\omega \leq 1$. An argument type annotated with 1 can accept a $\omega$, but not vice versa. In our situation, to use the running example, the ML type of the function `func4` matches that of the type specified by `foldr` while the *o*-type does not. Thus, by promoting the value `func4` up the ordering relation, we produce a coercion.

Interpreting *o*-types in this way presents a few problems. Subtyping is usually used to interpret a one-way notion of "computability", which is not the case here. By computability, we mean that some expression $\Gamma \vdash e_1 : \alpha$ can be used where a $\Gamma \vdash e_2 : \beta$ is expected, if $\beta \leq \alpha \in \Gamma$. For example, the operational constraints $\kappa \leq \rho$ and $\rho \leq \kappa$ are equally valid (if they are ML equivalent) in the sense that a formal argument typed as the latter (respectively former) must be able to accept arguments typed as the former (respectively latter). This is because, unlike the usual subtype ordering in which there is some restriction in computability, function application is valid for both types.

However, subtype semantics give us useful intuitive notions we can exploit readily. As we mentioned earlier, subtyping is treated by the CLI using subset semantics. This is usually extended by using a partial equivalence relation, or per, to build appropriate quotient sets. The per relation is used to augment the classic subset semantics because they provide an approximate sense of equivalence. This is because terms of type $Integer \rightarrow Integer$ and $Real \rightarrow Real$ are interpreted as both belonging to the same set of type $\tau \rightarrow \tau$. An alternative model is the *conversion* interpretation.

The conversion interpretation is designed around the observation that along with the usual set of values for each type in the language, there is an associated conversion function from the type $\tau$ to $\epsilon$, where $\tau \leq \epsilon$. We can model the subset semantics described above as a special case by defining conversion functions like *noop* to convert a term from type $\tau$ to $\epsilon$.

In $M_{CM}$, we borrow ideas from both interpretations. We define two partial equivalence relations, denoted $\sim$ and $\simeq$, representing a "weaker" and "stronger" sense of equivalence respectively over $o$-types.

We define the relation $\sim$ by using the ML arity of the type, that is, $\sim \equiv arity(\tau) = arity(\sigma)$. The quotient set over $\mathcal{ST}$ is constructed in the usual way and denoted $\mathcal{ST}/\sim$.

The equivalence class $[\tau]_\sim$ is generated in the usual way from $\{\tau' | \tau' \sim \tau\}$. In the case of the quotient set $\mathcal{ST}/\sim$, we can also order it intuitively over the natural number relation $\leq$. This is a partial order. For example, in Listing 1.1 we described the function `zipWith` of type $\tau$, which will only accept for its first argument functions with arity 2 or greater. This set is characterised by the principle ideal $\uparrow$`arity(head(`$\tau$`))`.

Homomorphisms between equivalence classes constitute our operational coercions, or dopes. We denote coercions by coercing from the type $\kappa$ to $\tau$ (respectively $\tau$ to $\kappa$) using the syntax $c^\tau_\kappa$ (respectively $c^\kappa_\tau$). Two $o$-types are equivalent if they are coerced to the same equivalence class. Much like the subtyping relation, we coerce over non-ground types using the usual rules of operational decomposition, except in the case of $\rightarrow$ types, in which there is no equivalent notion of contra/-covariance. If we apply a coercion over two types in the same equivalence class, then we assume that the operation translates to a noop.

We define $\simeq$ by defining two terms as strongly equivalent if they have the same syntactic $o$-type. Thus in section 5.5, the functions `func4` and `func5` are in the same equivalence class in the ordered set over $\sim$, but they are not in the stronger ordered set over $\simeq$.

For example, given the types $Integer \rightarrow \{Integer, Integer\}$ and $Integer \rightarrow Integer \rightarrow Integer$, they are in the same equivalence class in $\mathcal{ST}/\sim$, but are in different equivalence classes in $\mathcal{ST}/\simeq$. Therefore, if we were to coerce when compiling to eval/apply, it would resolve to a lambda dope, as opposed to a noop in push/enter.

## *7.5 Tail-End : Mapping Types to the CLR*

While this is strictly a Tail-End transformation, we discuss it here because it deals with transforming the type language into a format compatible with the two runtime targets, the $CLR_{\leq}$ and $CLR_{\leq,\forall}$. In this section, we look at the representation of values of the following types for the $CLR_{\leq}$ and $CLR_{\leq,\forall}$:

1. Value types
2. Type Schemes
3. Function types
4. Sum types

For each case, we look at how they are represented on the $\text{CLR}_{\leq}$ and $\text{CLR}_{\leq,\forall}$. In the case of function types, we have to look at how functions are encoded on the CLI and how they are treated as first class values. We do not need to consider product types separately from sum types because we encode the former in terms of the latter.

We assume, at this point, that we have an inference algorithm which assigns *o*-types to terms. The value types which we gave in Table 2.1.2 may be erased or modified in some way before they are compiled to the respective machines. We will discuss where this occurs for each output target.

## *7.6 Compiling Types to CLR$_{\leq}$*

Subtype polymorphism is a restricted form of parametric polymorphism. Unlike parametric type variables, which can range over any type, type variables on a subtype polymorphic system can only range over types related to each other by a subtype relation. However, this does not limit us, because in the $\text{CLR}_{\leq}$ all types are subtypes of the universal $\top$ type, *Object*.

### *7.6.1 Function Types*

In this section, we discuss how functions are compiled to the $\text{CLR}_{\leq}$, and how they are treated as first class values on the $\text{CLR}_{\leq}$.

A function in Mondrian is compiled to a `class` type, discussed in section 4.2.1. A function is instantiated and placed in the heap denoted as a FUN object.

#### *Functions as First Class Values*

There are two methods we can use to treat functions as first class values. The first is obvious, to instantiate the function from its class definition, and use *pass by reference*. To invoke a value of function type we use the class' `ENTER` method.

The second method uses a delegate to generate a function binding to the `ENTER` method. To invoke a formal argument of function type, we use the `(...)` syntax.

We do not need to specify the method name because the delegate is already bound to it. However, we still use the `ENTER(...)` syntax to invoke a value of function type that is free because it is not treated as a first class value.

We call this *pass by delegate*, although, strictly speaking, it is also an instance of pass by reference. However, we keep the distinction because in the former we have a reference to the class definition, while in the latter, we hold an indirection, which we pass by reference to the `ENTER` method of the class definition. Delegates on the $\text{CLR}_{\leq}$ are name equivalent, as opposed to structurally equivalent, and, as such, are not equivalent up to $\alpha$-reduction.

Passing by reference is operationally cheaper than pass by delegate, due to the way in which the CLI performs the function call to `ENTER`. Pass by delegate is slower, because invoking a delegate may incur the overhead of binding checks to the type (when passing in a string based method name) checks for the `Invoke()` member, and the cost of newing up an object array with the relevant arguments. We never bind to a string name so this cost is eliminated. However, unlike pass by reference, it does not require knowledge of the implementation type of the function when we want to use it.

Consider the following example:

```
// f : forall a, b => a -> e -> f;
f = x -> let // f2 : forall d, e, f => d -> e -> f;
             f2 = y -> z -> ..something..;
             // f3 : forall d, e, f => d -> e -> f;
             f3 = t -> s -> ..something..;
         in  (f2 x);

main = let // x2 : Integer;
           x2 = f 2 2;
       in  ...
```

Inferred types are annotated using comments as usual. The functions `f2` and `f3` are compiled to their respective classes, in this case, named after their variable bindings.

Consider the application `f 2 2` in the `let` expression of `main`. To compile this expression, we use its instantiated $o$-type, given as:

$Integer \rightarrow \{Integer, Integer\}$

Applying the rule $\text{EXACT}_n$ to the application creates the call `f 2` and returns the call continuation, which is an instance of a function of type $Integer \rightarrow Integer$.

However, to complete the function call, we have to know the implementation type. This is because the CLI requires that the result of function call be "named", either by a cast or by assigning to a variable of the appropriate type (which may also require a cast) before continuing the function call.

This is not so trivial, in this case, because `f2` is partially applied in `f`. Lambda doping would have generated a Closure whose implementation type would not generally be visible to a consumer of `f`. However, its *o*-type is visible, which in this case, is $Integer \rightarrow Integer$.

In general, we cannot rely on the fact that the implementation type of a function is known. Consider this example:

```
//f1 : {Integer, Integer} -> {Integer, Integer};
f1 = x -> let // x1 : Integer -> {Integer, Integer};
              x1 = y -> ....x...;

          in  x1 2;

//f2 : Integer -> {Integer, Integer};
f2 = a -> let // x2 : Integer -> Integer;
              x2 = y -> ...a..y.;

              // x3 : Integer -> Integer;
              x3 = x -> ...a..x.;

          in  f1 x2;
```

The application `f1 x2` returns a function of type $Integer \rightarrow Integer$ whose implementation type we do not know. This begs the question of how to cast the return type of `ENTER`. One way is not to cast at all by "erasing" the return type to *Object*. This, however, just moves the problem to an application call using `f2`. For example, in the application `f2 1 2`, the call `f 1` returns a function which is typed as *Object*. To invoke `ENTER` on the call continuation, we have to cast the value to the appropriate implementation type, which, as previously mentioned, was unknown. In this case, this is not strictly true, but to find out would require us to analyse the behaviour of `f1` and note the implementation type names assigned during compilation. This is a futile exercise if the function definition is large.

This suggests we need to be able to return the function in such a way as to enable us to name the implementation type of the value returned.

There are two easy solutions:

- adopt a naming scheme given by the function $\mho$ that, when given a $o$-type, generates a unique name, up to $\alpha$-reduction for the implementation type

- use a name invariant indirection, such as a delegate, which, when invoked, calls the appropriate implementations' `ENTER` method.

The first solution suffers from the problem that different let bindings with the same $o$-types will generate the same name, leading to name clashes. A delegate is a good solution, because for a delegate of a specific $o$-type we can generate a unique name. Using the previous example, the application `f1 x2`, with $o$-type $Integer \rightarrow Integer$ would return a instance of a delegate declared in $C^\sharp$ as:

```
delegate Integer D_1 (Integer x)
```

We will define some notational conventions before we continue. We will adopt a naming function $\mho$ for delegates, typed as $\kappa \rightarrow String$. It prepends the prefix `D_` to a string $s$, where $s$ is a unique identifier. The usual format for $s$ is a integer encoding the arity of the delegate. However, this only works if we are generating a single delegate per equivalence class in $\mathcal{ST}/\simeq$, because, if we have two delegates with the same arity but different argument/return types, then we could not uniquely identify them using a naming scheme based solely on arity. This is discussed in more detail in the next section.

We assume that there is a set of delegates $\mathcal{D}$ indexed by a set of strings $S$, which are generated by $\mho$. Given a $o$-type $\kappa$, if $\mho(\kappa) \in \mathcal{D}$ then $\mho(\kappa)$ will return the unique string. We can extend the set $\mathcal{D}$ using $\mathcal{D}, s : \sigma$ where $s$ is a string and $\sigma$ its $o$-type.

A delegate can also take another delegate as a parameter or return a delegate following the usual higher order reduction semantics. For example, if we were to replace the expression `f1 x2` with `f1`, then the delegate returned would be defined as:

```
delegate D_1 D_2 (D_1 a)
```

Intuitively, this delegate is taking a function which is taking a function of type $Integer \rightarrow Integer$ and returning a function of type $Integer \rightarrow Integer$. The `ENTER` function for `f1` is defined as:

```
D_1 ENTER (D_1 a)
```

*Number of Delegates on the CLR$_{\leq}$*

The number of delegate definitions generated depends on how many function types we have per function arity. If we generated a new delegate for every combination of base types, then we would have a combinatorial explosion. For example, given a type system with only three base types for the worst case, we would have to generate $3^N$ different delegate types, of arity $N$. This shows it is of exponential complexity.

On the CLR$_{\leq}$, we can reduce this complexity by erasing types to a single base type, in this case, *Object*, to which every type can be promoted. This reduces the combinations to linear complexity, because we only need one delegate to represent a function of any arity $N$. We only need a single delegate per equivalence class in $\mathcal{ST}/\simeq$. This means our naming convention, given by $\mho$, is valid, and will generate a uniquely identifiable delegate given its *o*-type. Note that we cannot parameterise the delegate because CLR$_{\leq}$ does not support generics. This is discussed in section 7.7.2.

However, this puts further restrictions on the method definition `ENTER`. Using this method, we now require that the `ENTER` function for `f1` be declared as:

```
Object ENTER (Object x)
```

Consider the type:

$(\alpha \to \beta) \to \alpha \to \beta$

After type erasure it becomes:

$Object \to Object \to Object$

This can be encoded as a delegate using the declaration:

```
delegate Object D_2 (Object a, Object b)
```

Notice that we have erased the function type $\alpha \to \beta$ to *Object*. Erasing function argument types requires us to downcast the value when we want to use it. In theory, this is not a problem, because we have checked statically that all applications are valid. However, this is only valid inside Mondrian programs and does not guarantee that a client using the code will supply values of the right type. These malformed programs will only be caught at runtime. We accept this as a cost of this compilation scheme.

We can now establish an correspondence between an *o*-type and its translation to the CLR$_{\leq}$, given in Table 7.1.

| $o$-type | delegate definition |
|---|---|
| $\alpha_1 \rightarrow \ldots \alpha_n$ | `delegate` $Obj_n$ `D_`$(n-1)$`(`$Obj_1$ `x`$_1 \ldots Obj_{n-1}$ `x`$_{n-1}$`)` |
| $\alpha_1 \rightarrow_{k-1} \{\beta_1, \ldots \beta_n\}$ | `delegate` $Obj$ `D_`$(k-1)$ `(`$Obj_1$ `x`$_1 \ldots Obj_{k-1}$ `x`$_{k-1}$`)` |

Table 7.1: Encoding from $o$-types and delegates on CLR$_{\leq}$, where $Obj$ represents $Object$

### *7.6.2 Type Schemes*

We erase all parametric type variables to type *Object*. In essence, we are merely introducing a upper bound for quantified variables of type *Object*. This converts M$_{\leq,\forall}$ terms to $M_{\leq}$ terms. Unfortunately, this also means we remove the static type safety when interfacing with consumers, as we can supply a value of any type.

Given the type signature

$\forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow List < \alpha > \rightarrow List < \beta >$

inferred for `map`, we type erase this to:

$Object \rightarrow List \rightarrow List$

### *7.6.3 Sum Types*

The CLR$_{\leq}$ does not allow parameterised types, therefore, we erase all parameterised types to their non-parameterised forms. Fields typed using type variables are typed using *Object*. The `List` data type defined parametrically in Mondrian as

```
class List<A> {};
class Nill<A> : List <A> {};
class Cons<A> : List <A> { head : A; tail : List<A>};
```

is erased to:

```
class List {};
class Nill : List {};
class Cons : List { head : Object; tail : List};
```

### *7.6.4 Product Types*

The Product type is trivially erased to:

```
class Pair
{   fst : Object;
;   snd : Object;
};
```

Listing 7.4: Erased Pair type

### *7.6.5 Type Erasure*

We are now in a position to define a type erasure algorithm which will provide the right type information to compile values of value, function and sum type. We define a type erasure algorithm that follows the following guidelines:

1. Type variables are erased to *Object*s
2. Type schemes are erased by removing type variables
3. Parameterised types are erased to their equivalent "base" type
4. Function types are erased to *Object*

**Definition 7.6.1.** *We proceed by structural induction over functional $M_{CM}$ terms. Extending the algorithm to all $M_{CM}$ terms is trivial. The function* `erasemonsub` *erases types by induction over terms. As usual, top-level functions are treated as recursively defined let expressions. For the sake of brevity, we assume that they are treated that way, and that expression e, which has been denoted with a prime as $e'$, is equivalent to $e' = etype_{M_\leq}(e)$.*

$$
\begin{aligned}
&eterm_{M_\leq}(x) \;=\; etype_{M_\leq}(x)\\
&eterm_{M_\leq}(\lambda x.t_1) : \tau_1 \;=\; (\lambda eterm_{M_\leq}(x).eterm_{M_\leq}(t_1)) : etype_M(\tau_1)\\
&eterm_{M_\leq}(t_1\ t_2) = eterm_{M_\leq}(t_1')\ eterm_{M_\leq}(t_2')\\
&eterm_{M_\leq}(let\ x\ =\ t_1\ in\ t_2)\\
&\qquad\qquad =\ let\ x\ =\ eterm_{M_\leq}(t_1')\ in\ eterm_{M_\leq}(t_2')
\end{aligned}
$$

$$
\begin{aligned}
&etype_{M_\leq}(\tau) \;=\; Object\\
&etype_{M_\leq}(v) \;=\; Object\\
&etype_{M_\leq}(T)\ \overline{\tau_1} \;=\; T\\
&etype_{M_\leq}(\tau_1 \rightarrow \tau_2) \;=\; Object\\
&etype_{M_\leq}(\forall \overrightarrow{\alpha}.\sigma) \;=\; etype_{M_\leq}(\sigma)
\end{aligned}
$$

For example, we can write:

```
f : forall a, b => (a -> b) -> b;
f g = let //f2 : forall c => b -> c;
          f2 = x -> g x;
          //f3 : forall a, b => (a -> b) -> b;
          f3 = p -> z -> p z;
      in  f3 (f2 2) 2;
```

The inferred types are given as comments. The erased typings are given as:

```
f : Object -> Object;
f g = let f2 : Object -> Object;
          f2 = x -> g x;
          f3 : Object -> Object;
          f3 = p -> z -> p z;
      in  (f3 (f2 2) 2;) : Object
```

## *7.7 Compiling Types to CLR$_{\leq,\forall}$*

The CLR$_{\leq,\forall}$ gives us an explicit way of expressing parametric polymorphism through generics. Generics, like parametric polymorphism, allows us to explicitly quantify over type variables.

### *7.7.1 Using Generics*

The CLR$_{\leq,\forall}$ requires the explicit application of types to functions when using generic classes. A generic class is defined in the following way:

```
class Test<A, B>
{
    A Foo (B b)
    {   ...;
    }

}
```

The type variables `A` and `B` parameterize the type `Test`, which are visible to all definitions within the `class` declaration.

When the runtime requires a particular instantiation of parameterised class, the loader checks to see if the instantiation is compatible with any that it has seen before. If not, then a field layout, or *vtable*, is determined and a new value

is created which is to be shared between all compatible instantiations. The items in this vtable are the entry stubs for the methods of the class. When these stubs are later invoked, they will generate just in time (JIT) code to be shared for all compatible instantiations. When compiling the invocation of a (non-virtual) polymorphic method at a instantiation, we first check to see if we have compiled such a call before for some compatible instantiation. If not, then an entry stub is generated, which will in turn generate code to be shared for all compatible instantiations.

In summary, this means we have to instantiate explicitly a new instance of a polymorphic function whenever we wish to use it. In the case of the class `Test`, we use the syntax:

```
Test<Integer, Integer> a = new Test<Integer, Integer>;
```

In this case, the variables `A` and `B` are instantiated to `Integer` and `Integer`.

### *7.7.2 Function Types*

We have discussed in section 7.6.1 how functions are compiled and treated as first class values. For the most part, the same reasoning applies to functions on the $\text{CLR}_{\leq,\forall}$. That is, we compile functions to class definitions and use delegates to treat functions as first class values. However, we do not need to erase types because generics lends us a form of structural equivalence. That is, we can use the same generic delegate, defined as

```
delegate A SomeDelegate<A, B> (B a)
```

to represent functions of type $Integer \rightarrow Real$ and $Real \rightarrow Integer$. This gives us linear complexity over $N$ (the arity of the function) without requiring type erasure. This is equivalent to generating a single delegate per equivalence class in $\mathcal{ST}/\simeq$.

We can summarise the equivalence between an $o$-type and a delegate encoded on $\text{CLR}_{\leq,\forall}$ in much the same way we did earlier, given in Table 7.2.

### *7.7.3 Type Schemes*

Type schemes can be directly represented on the $\text{CLR}_{\leq,\forall}$, since type operators can be quantified in the class definition. We can translate the type

$\forall\alpha\beta.(\alpha \rightarrow \beta) \rightarrow List < \alpha > \rightarrow List < \beta >$

to:

| $o$-type | delegate definition |
|---|---|
| $\alpha_1 \rightarrow \ldots \alpha_n$ | `delegate` $\alpha_n$ `D_`$(n-1)$`<`$\alpha_1, \ldots \alpha_k$`>` `(`$\alpha_1$ `x`$_1, \ldots \alpha_{n-1}$ `x`$_{n-1}$`)` |
| $\alpha_1 \rightarrow_{k-1} \{\beta_1, \ldots \beta_n\}$ | `delegate D_`$(n-1)$ `D_`$(k-1)$`<D_`$(n-1)$`,` $\alpha_1, \ldots \alpha_{k-1}$`>` `(`$\alpha_1$ `x`$_1, \ldots \alpha_{k-1}$ `x`$_{k-1}$`)` |

Table 7.2: Encoding from $o$-types and delegates on $\text{CLR}_{\leq,\forall}$

```
class Test <A, B>
{
    List <B> ENTER (D_1<A, B> a, List <A> b)
    {   ...
    }
}
```

### *7.7.4 Sum types*

Quantified sum types are again directly encoded as `class` declarations in $C^\sharp$. The `BinaryTree` data type defined in Listing 2.2 is compiled to the generic classes given in Listing 7.2.

### *7.7.5 Product Types*

The Pair can be trivially translated to $\text{CLR}_{\leq,\forall}$ in the following way:

```
class Pair<A, B>
{   A : fst;
    B : snd;
};
```

Listing 7.5: Pair defined in $\text{M}_{\leq,\forall}$

## *7.8 Type Inference Rules*

We now define the inference rules for type assignment in Mondrian. Note that we restrict coercions for $o$-types over the rules TA-ABS and TA-NEW. We also restrict subtyping to TA-APP, TA-IF, TA-CASE and TA-NEW to give us syntax-directed inference rules. We do not keep track of fresh variable names explicitly, but assume there is a name source that generates unique type variables where required.

### *7.8.1 Declarations*

**Definition 7.8.1.** *Syntax-directed type rules for variable declarations.*

$$\frac{\Gamma \vdash T_1 \qquad \Gamma, \overline{\alpha_i} \vdash \overline{x_i : \tau_i}}{\Gamma \vdash (class\ T\ \overline{\alpha_i}\ T_1\ \overline{x_i : \tau_i}) : T}\ TA - CLASS$$

$$\frac{\Gamma \vdash\ v\ =\ e : \tau}{}\ TA - VARDECL$$

### *7.8.2 Functional*

**Definition 7.8.2.** *Syntax-directed type inference rules for functional $M_{CM}$.*

$$\frac{\Gamma \vdash t : \tau \vdash t : \tau}{}\ TA - VAR$$

$$\frac{x : \alpha, \Gamma \vdash y : \beta}{\Gamma \vdash (\lambda x.y) : \alpha \rightarrow\ oannotate(\beta)}\ TA - ABS$$

$$\frac{\Gamma \vdash f : \alpha \rightarrow \beta \qquad \Gamma \vdash x : \phi \qquad \Gamma \vdash \phi \leq \alpha}{\Gamma \vdash f\ x : \beta}\ TA - APP$$

$$\frac{\Gamma \vdash e' : Close(\sigma, (\Gamma, x : \sigma)) \quad \Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash (let\ x = e'\ in\ e) : \tau}\ TA - LET$$

$$\begin{aligned} \text{Close } (\tau, \Gamma) \quad &\equiv\ \forall \alpha_1, \ldots, \alpha_n.\tau \\ &\text{where } \alpha_1, \ldots, \alpha_n = FTV(\tau)/FTV(\Gamma) \end{aligned}$$

The functional rules are consistent with the literature for the most part. We use the function `Close` to generate quantified let declarations.

The rule TA-ABS uses the function `oannotate` to annotate the return type variable of the function. We explain why this is done later when we discuss inferring operational types, in section 7.9.1. The rule also uses the primitive subsumption relation $\leq$. Therefore, we do not use the usual rule $\alpha \leqslant \beta \rightarrow \gamma$, but a more restricted version, $\beta \leq \alpha$. Note that we have to reverse explicitly the constraint because we cannot rely on the usual structural decomposition rules for $\rightarrow$ types. We have modified the rule T-APP to include type subsumption, this structures subsumption over TA-APP.

### *7.8.3 Simple Extensions*

The rules for simple extensions are given in Definition 7.8.3. TA-IF is typed as the lub of the two branches. If a lub does not exist, then we have a type error. We have two type rules for `switch` which are used according to the type of the scrutinee. If the type is a reference type, rule TA-CASE, then the type of the scrutinee type is the lub of the case types. If the scrutinee is a value type, rule TA-SWITCHVAL, then the scrutinee and case branches are of the same type, since value types are sealed in the CLI.

**Definition 7.8.3.** *Syntax-directed type inference rules for $M_{CM}$ extensions.*

$$\frac{\Gamma \vdash t_1 : bool \quad \Gamma \vdash t_2 : \tau_1 \quad \Gamma \vdash t_3 : \tau_2 \quad \tau_0 = \tau_1 \vee \tau_2}{\Gamma \vdash (if\ t_1\ then\ t_2\ else\ t_3) : \tau_0}\ TA-IF$$

$$\frac{class\ T\ \overline{\alpha_k}\ =\ \overline{C_i\ \overline{\tau_{ij}}} \quad \Gamma \vdash\ t_0 : C\ \overline{\tau_j} \quad \Gamma \vdash\ \bigvee C_i \leq\ C \quad foreach\ i\ \Gamma, \overline{x_{ij}} : \tau_{ij} \vdash e_i : \tau_i \quad \Gamma \vdash e_{def} : \tau_{def} \quad \Gamma \vdash \bigvee \tau_i \vee \tau_{def} \leq \tau}{\Gamma \vdash\ (switch\ t_0\ of\ \overline{(case\ C_i\ \overline{(x_{ij}\ =\ e_{ij})}\ :\ e_i : \tau_i)}\ e_{def} : \tau_{def}) : \tau}\ TA-CASE$$

$$\frac{\Gamma \vdash\ t_0 : \iota \quad \Gamma \vdash \overline{v_i : \iota_i} \quad foreach\ i\ \ \Gamma, v_i : \iota_i \vdash e_i : \tau_i \quad \Gamma \vdash e_{def} : \tau_{def} \quad \Gamma \vdash \bigvee \tau_i \vee \tau_{def} \leq \tau}{\Gamma \vdash\ (switch\ t_0\ of\ (\overline{case\ v_i\ :\ e_i : \tau_i})\ e_{def} : \tau_{def}) : \tau}\ TA-SWITCHVAL$$

$$\frac{class\ T\ \overline{\alpha_k}\ =\ \overline{C_i\ \overline{\tau_{ij}}} \qquad \Gamma, v : C\overline{\tau_j}, \overline{\alpha_k} \vdash \overline{e_j : \kappa_j}}{\Gamma \vdash\ new\ v : C\ \overline{\alpha_k}}\ TA-NEW$$

## *7.9 The Inference Algorithm*

The inference algorithm is derived from the syntax directed type rules for $M_{CM}$. We base our type system on the one defined by Damas & Milner in (Damas & Milner 1982) extending it to infer *o*-types. The ML typing algorithm gives us "simple" polymorphic type inference, that is, polymorphism without constraints. We define the following decision procedures:

- `subtype(`$\Gamma$`,` $\phi$`,` $\tau$`)` : checks that we can coerce from type $\phi$ to type $\tau$ in the environment $\Gamma$

- `unify(`$\Gamma$`,` $C$`)` : checks that we can unify constraints
- `otype(`$\Gamma$`,` $\tau$`)` : returns an operational type from the type $\tau$
- `infer(`$\Gamma$`,` $e$`)` : infers the principle otype for the expression.

**Definition 7.1.** *The subtype decision procedure is derived from the subtype type rules.*

*Two types are equivalent if there exists a constraint in the environment or if we are coercing to Object. The constraint must exist in the environment* $\Gamma$*. The types to compare are given as* $\phi$ *and* $\tau$*. The* `subtype` *procedure assumes that constructed types are decomposed by* `unify`*.*

```
subtype (Γ, φ, τ)
    = if τ == Object,
      then true
      else if φ == φ1 → φ2 and
              τ == τ1 → τ2
           then fail
           else if φ ≤ τ ∈ Γ
                then true
                else false
           else fail
```

Listing 7.6: Subtype algorithm

**Definition 7.2.** *The unify algorithm is given in Listing 7.7. Constraints are equations between o-types and types and are stored in the constraint set* $C$*. We unify by taking the expression* $\Gamma$ *and the constraint set* $C$ *and picking a constraint non-deterministically. We extend the substitution set* $S$ *by binding variables to types.*

The unification algorithm is for the most part, consistent with that given by Milner.

We have added a condition for unifying two annotated types, given in lines 23-24, by unwinding them and unifying the components individually. If we are attempting to unify two ground types that are not name equivalent, then we check that there is a lub, given in line 25.

```
unify(Γ, S, C)
 = if C == {} then S
   else let {φ = τ} ∪ C' = C
        in  if φ == α and α ∉ FV(τ)
            then if S(φ) == φ
                 then ([α ↦ τ] ∘ S)
                 else unify (Γ, S, [S(φ) = τ]C)
            else if τ == α and α ∉ FV(φ)
                 then if S(τ) == τ
                      then ([α ↦ φ] ∘ S)
                      else unify (Γ, S, [S(τ) = φ]C)
                 else fail
            else if φ == (φ1 → φ2) and τ == (τ1 → τ2)
                 then unify (Γ, S, {φ1 = τ1, φ2 = τ2} ∪ C')
            else if φ == {φ1 ... φn} and
                    τ == {φ1 ... τn}
                 then let φ' = expandotype φ
                          τ' = expandotype τ
                      in  unify (Γ, S, {φ' = τ'} ∪ C')
            else if φ == N and τ == N2
                 then if N == N2 or subtype (Γ, N, N2)
                 then unify(Γ, S, C')
                 else fail
            else if φ == N<φ1 ... φn> and
                    τ == N2<τ1 ... τn>
                 then if N == N2 or (subtype (Γ, N, N2))
                      then unify (Γ, S, {φ1 = τ1, ..., φn = τn} ∪ C')
                      else fail
                 else fail
            else fail
```

Listing 7.7: Unify algorithm

### *7.9.1 Generating Operational Types*

Operational typings can be generated by a "structured" application of the substitution list to the type generated by the inference algorithm. We define the function `otype`, which is typed as $(S, \tau) \rightarrow \kappa$, where $S$ is some substitution and $\tau$ some type. It applies the substitution to the type $\tau$ producing an $o$-type $\kappa$. Intuitively, we are treating the set of substitutions as a record of how applications have been assembled. By traversing the set of substitutions, and using the annotation $\{\}$ to distinguish the return type in an abstraction, we can build up a record of reduction.

Before we give the algorithm for `otype`, we build up a understanding of how it works by example:

```
f = x -> let g = y -> y; in g;
```

The $o$-type inferred for `f` is:

$\forall\alpha\beta.\alpha \rightarrow \{\beta, \{\beta\}\}$

We generate this type by first typing the let binding using the T-VAR rule which types the body in the environment $\Gamma = f : 1$. Rule TA-ABS types the lambda abstraction, typing the body, which is a `let` expression in the environment $\Gamma = f : 1, x : 2$. The type of `let` is inferred as $4 \rightarrow \{4\}$. Finally, TA-ABS types the lambda as $1 \rightarrow 3$, returning the substitution set $S$, given as:

```
{ 1 ↦ (2 → {3}) ; 3 ↦ (4 → {4}) }
```

We can now generate a type for the application `f` by using the function `otype(S, 1)`, which returns the type:

$2 \rightarrow \{4, \{4\}\}$

This can be quantified in the usual way, giving us $\forall\alpha\beta.\alpha \rightarrow \{\beta, \{\beta\}\}$, after $\alpha$-reduction. The types 3 and 4 are given the annotations $\{3\}$ and $\{4\}$ to indicate that this is the return type of the function. `otype` will wind the type substituted for 3. In this case this is $4 \rightarrow \{4\}$, which becomes $\{4, \{4\}\}$. We can see here that the annotation $\{\}$ has indicated the "end" of the type $2 \rightarrow 3$. Note that in this example, we can drop the last annotation of the type variable 4 because it will not change the reduction semantics for a value typed as this.

In general, when we apply a substitution to a type variable in a contravariant position, we wind it, if it is a function. The only time we wind a type substitution to a type variable in a covariant position is when it is annotated with $\{\}$. Substitutions are applied by traversing the substitution tree in a depth first manner. This

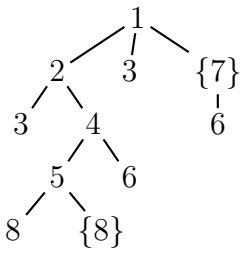


Figure 7.3: Representing a substitution list as a tree

ensures that we do not wind a function type prematurely, that is, before it is fully substituted. Consider the following substitution set:

```
{ 1 ↦ (2 → 3) ; 2 ↦ (5 → 6) ; 6 ↦ (7 → 8) }
```

If we wind the substitution for the type variable 2 before we reconstruct the type, then we would end up with type variable 2 replaced with $\{5, \{7, 8\}\}$, instead of $\{5, 7, 8\}$, which is incorrect. Note that if type variable 6 was annotated with $\{6\}$, then it would be a correct type.

A more complicated example, which shows how the function traverses the substitutions in a depth first manner, is given below:

```
let p = c -> x -> c x l;
    l = y -> y;
in  ...
```

We type `p` using the rules TA-VAR and TA-ABS. The body of the abstraction is typed in the environment $\Gamma = p : 1, c : 2, x : 3$. `infer` returns the substitution set $S$, given as:

```
{ 1 ↦ (2 → 3 → {7}) ; 2 ↦ (3 → 4) ; 4 ↦ (5 → 6) ; {7} ↦ 6 }
```

Before applying the substitutions, we infer the type of `l` in the environment $\Gamma = p : 1, l : 5$, which returns the following additions to the substitution set $S'$:

```
{ ... ; 5 ↦ (8 → {8}) ; ... }
```

To aid in visualisation, consider Figure 7.3, which shows the graph of the substitution set. It is easy to see that an *o*-type construction is a simple depth first collapse, using the function call `otype(Γ, 1)` to build the type of `p`. When `otype` applies the substitutions, it will wind the substitution to type variable 5, because the type variable appears in a contravariant position. Thus, `otype` returns the *o*-type:

$\{3, \{8, \{8\}\}, 6\} \rightarrow 3 \rightarrow \{6\}$

Note that type 1 is not considered contravariant nor annotated, thus we do not wind the substitution for 1, to avoid the following:

$\{\{3, \{8, \{8\}\}, 6\}, 3, \{6\}\}$

Finally, after applying `close` to the type of `p` and `l`, we get the quantified types:

$\forall\alpha\beta\tau.\{\alpha, \{\beta, \{\beta\}\}, \tau\} \rightarrow \alpha \rightarrow \{\tau\}$
$\forall\alpha.\alpha \rightarrow \alpha$

Consider the following application using the function `p`:

```
i = a -> b -> a + b;
g = a -> b -> b;
h = p g i;
```

The application `(p g)` returns a function of type $\{8, \{8\}\}$, using the same type variables from the previous example. Applying this function to `i` produces a substitution unifying 8 to $Integer \rightarrow Integer$. The `otype` algorithm will wind the application of the substitution to $\{8\}$, giving us the type $\{Integer, Integer\}$ for the application `p g i`. This is correct, hence we unwind the type to apply it to `2` with an application of the $\text{EXACT}_n$ rule.

### *7.9.2 Otype Algorithm*

**Definition 7.3.** *The simplified `otype` algorithm is defined in Listing 7.8. We use the usual substitution syntax $S(x)$ to describe a single instance of the application of a substitution to a type variable $x$. The function `mapType` is defined first. It is a three argument function, taking the type $\tau$, and two functions `f1` and `f2` of type $\tau \rightarrow \kappa$. The function `otype` takes a type $\tau$, and traverses it depth-first using the substitution list $S$, effectively treating it as a tree. As it returns up the substitution tree, it collapses the type using the function `flattenfunc`, which removes the $\rightarrow$ operator and replaces it with the cons operator `::`. The expression `otype(S, otype(S, τ))` will return the same type, that is, the o-type is the fixpoint of the function `otype` provided we use the same substitution set $S$. We simplify the function `mapType` so that we only consider traversing types constructed with $\rightarrow$. Extending `mapType` to traverse other type constructors is trivial. In line 14, the surrounding brackets guarantee that the type returned by the function call `{ f1 (τ') }` is annotated.*

```
mapType(τ, f1, f2)
    = if τ == τ1 → τ2
      then let τ1' = f1 (τ1)
               τ2' = mapType (τ2, f1, f2)
           in  τ1' → τ2'
      else f2 (τ)

otype(S, τ)
    = let f1 (τ) = let τ' = otype (S, S(τ))
                       τ'' = flattenfunc τ'
                   in  τ''

          f2 (τ) = if τ == {τ'}
                   then { f1 (τ') }
                   else if τ == S(τ)
                        then τ
                        else otype(S, S(τ))
      in  mapType (τ, f1, f2)

flattenfunc (τ)
    = if τ == τ1 → τ2
      then τ1 :: flattenfunc τ2
      else τ
```

Listing 7.8: Otype algorithm

```
otypeannotate (S, e:τ)
  if e == x
       return e:otype (S, τ)
  else if e == (λx.e1)
       let (e1') = otypeannotate (S, e1)
       in  return (λx.e1'):otype (S, τ)
  else if e == e1:τ1 e2:τ2
       let (e1') = otypeannotate (S, e1)
           (e2') = otypeannotate (S, e2)
       in  return (e1' e2')
  else if e == let x = e1 in e2
       let (e1') = otypeannotate (S, e1)
           (e2') = otypeannotate (S, e2)
       in  return (let x = e1' in e2')
  else if e == (switch swe of {case C1:e1 ... Cn:en} default edef)
       let (swe') = otypeannotate (S, swe)
           (e_i') = forall i ∈ {1...n} otypeannotate (S, ei)
           (edef') = otypeannotate (S, edef)
           κ = sumotype e_i'
       in  return ((switch swe of {case C1:e1' ... Cn:en'}
                     default edef'):κ)
  else if e == (if e then e1 else e2)
       let (e1') = otypeannotate (S, e1)
           (e2') = otypeannotate (S, e2)
           κ = sumotype (e1, e2)
       in  return ((if e then e1 else e2):κ)
  else if e == Closure n ce e_i pend total
       then let (ce') = otypeannotate (S, ce)
            in  Closure n ce' {...} pend total
  else if e == New n {x1 = e1 ... xn = en)}
       then let e_i' = forall i ∈ {1...n} otypeannotate (S, ei)
            in  return (New n {x1 = e1' ... xn = en))
```

Listing 7.9: Otypeterms algorithm

We can apply `otype` over terms using a standard term induction using the function `otypeterms`, which applies `otype` to each type annotated term. `otypeterms` annotates `if` and `switch` expressions taking into account the points we raised in section 5.3.3. We annotate `if` and `switch` expressions using an $o$-type sum. This results in a an $o$-type with the smallest o-arity, that is, one requiring the most applications of $\text{EXACT}_n$. `otypeterms` is defined in Definition 7.4.

**Definition 7.4.** *The `otypeannotate` algorithm is given in Listing 7.9. We annotate terms by calling `otype` over the $M_{CM}$ language. The o-type assigned to conditionals is the most general o-type. We recall this is the o-type requiring the most number of applications of the rule* $\text{EXACT}_n$. *The function `sumotype` returns the sum an arbitrary number of expression's o-types. If there is more then one, then it will pick one arbitrarily. We use `type` to return the type assigned to an expression $e$.*

We can now complete the inference algorithm, using the `unify`, `subtype`, `otype` and `otypeannotate` decision procedures.

### *7.9.3 $M_{CM}$ Infer Algorithm*

**Definition 7.5.** *The inference algorithm defined in Listing 7.10 is generated from the syntax directed type rules for $M_{CM}$. We use the function `oannotate` to annotate types. We use $C$ for the constraint set, and $S$ for the set of substitutions. The function `flattenMap` takes a type with a top-level $\rightarrow$ constructor and generates an o-type by folding the type, using the $:$ operator and annotating it within a $\{\}$. For example, $\alpha \rightarrow \beta$ is converted to $\{\alpha, \beta\}$. We call `otypeterms` to annotate each term with the appropriate $o-type$ when inferring a `let` bound expression. Alternatively, we could have applied `otypeterms` after inference.*

```
infer (Γ, x)
          = (δ(S), δ(τ))
            where ∀α1 ... αk.τ where S = Γ(x)
                  δ = id[α1 ↦ α'1 ... αk ↦ α'k]
                  α1, ..., αn, are fresh variables


infer (Γ, e1 e2)
          = (S', α)
            where (S, τ1) = infer (Γ, e1)
                  (S', τ2) = infer (Γ, e2)
```

```
                S' = unify (Γ, τ1 = τ2 → α)
                α is a fresh variable


infer (Γ, λx.e)
        = (S' ∪ β' ↦ φ, α → β')
          where (S', φ) = infer (Γ ∪ {x : α}, e)
                β' = oannotate β
                α, β are fresh variables


infer (Γ, switch x of (alt_1...alt_n) default e_def)
        = (S', α)
          where (S, τ) = infer(Γ, x)
                (S1, τ1) = inferalt(Γ, alt_1)
                ...
                (Sn, τn) = inferalt(Γ, alt_n)
                (φ) = lub (Γ, τ1∪,...τn)
                (S') = unify (Γ, S ∪ Sn, α = φ, τ1 = φ, ..., τn = φ)
                α is a fresh variable


inferalt (Γ, C x1...xn => e)
        = (S, τ)
          where (S, τ) = infer (Γ ∪ {x1 : τ1, ..., xn : τn}, e)


infer (Γ, if e then e1 else e2)
        = (S''', α)
          where (S, τ) = infer (Γ, e)
                (S', τ1) = infer (Γ, e1)
                (S'', τ2) = infer (Γ, e2)

                (τ3) = lub (τ1 ∪ τ2, α)
                (S''') = unify (Γ, S'' ∪ S' ∪ S, α = τ3, τ1 = τ3, τ2 = τ3)
                α is a fresh variable


infer (Γ, let x = e'1 in e2 )
        = S' ∪ S'' κ'
          where (S, τ) = infer (Γ ∪ {x : α}, e1)
```

```
                    (S', τ') = infer (Γ ∪ {x : α}, e2)

                    κ = otype (S, τ)
                    e1' = otypeterms (S, e1)
                    (S'', κ') = close (Γ, S' ∪ S, κ)
                    α is a fresh variable


infer (Γ, new n e)
          = (S, τ)
            where (S, τ) = infer (Γ, e)
```

Listing 7.10: Infer algorithm

**Definiton 7.6.** *The lub of a set of types is the least upper bound in environment* $\Gamma$. *We define the functions* `lcp` *and* `lub` *in Listing 7.11.* `lcp` *returns the least common upper bound given two types* $\phi$ *and* $\tau$. *Recall that the subtype relation is reflexive, so, if we are comparing two types that are the same, this will return true. The function* `hasP` *checks if the type has a parent type, i.e., if it inherits from any other type. The function* `getP` *returns the parent type if it exists, which, in this case, will always succeed because we are checking beforehand.* `lub` *will return the lub of a set of types by calling* `lcp` *repeatedly. If there is no lub, for example, if we are attempting to find the lub of a set of type variables, then it will return the supplied type* $\alpha$.

```
lcp(Γ, φ, τ)
 = if φ ≤ τ ∈ Γ
   then τ
   else if hasP(Γ, φ) and hasP(Γ, τ)
        then lcp(Γ, getP(Γ, φ), getP(Γ, τ))
        else fail


lub(Γ, C, α)
 = if C == {} or C == {τ} then α
   else let {φ = τ} ∪ C' = C
        in  case lcp(Γ, φ, τ)
            {   fail    = α
            ;   τ' = lub (Γ, C', τ')
            }
```

Listing 7.11: Least upper bound (lub) algorithm

# Chapter VIII

# Tail-End : Compiling $\mathrm{M}_{CM}$ to the $s$-MSM

We discuss the compilation process to the machine $s$-MSM on the $\mathrm{CLR}_{\leq}$ and $\mathrm{CLR}_{\leq,\forall}$. The compilation model assumes that all function application is saturated, described in chapter 5, and that types have been properly formatted, described in section 7.5, for the two runtime targets.

## *8.1 $M_{CM}$ Translation*

A MSM program is a set of global declarations. These declarations can take the form of the `class` declarations, variable bindings and `import` statements.

**Definition 8.1.** *We define the function $\mathcal{C}$ which translates the Mondrian term in the environment $\rho$ to $C^\sharp$ syntax. We give each case by pattern matching on the appropriate term. We extend the environment $\rho$ using $\rho[x]$ where $x$ is some expression, usually a variable. We access the type of $x$ using $\rho(x)$. We use the "overline" notation $\overline{x_i}$ to indicate the process is repeated for each element $x$, in the list indexed by $i$. To ensure eager semantics, we reduce all let bound expressions when evaluating the body of the let. We indicate this by calling* `reduce` *for each let bound expression where* `reduce` *is defined in section 8.2.4.*

$\mathcal{C}\langle class\ n\ d\ \overline{v_i : \alpha_i}\rangle\rho$
```
     = Class C⟨n⟩ρ extends C⟨d⟩ρ
       {      x̄ᵢ = eᵢ
       }
```

$\mathcal{C}\langle import\ n\rangle\rho$ `= import n`

$\mathcal{C}\langle v = (\lambda\overline{x_i}.e)\rangle\rho$
```
     = let x̄ᵢ' = C⟨x̄ᵢ⟩ρ
           e'  = C⟨e⟩ρ[x̄ᵢ']
       in  Class C⟨v⟩ρ
           {     ρ(e) ENTER (x'ᵢ : ρ(x'ᵢ))
                 {    e';
```

```
                }
            }

C⟨v = e⟩ρ
    = let e' = C⟨e⟩ρ
      in  Class C⟨v⟩ρ
          {     ρ(e) ENTER ()
                {   e';
                }
          }

C⟨c⟩ρ = c

C⟨x⟩ρ = ρ(x)

C⟨switch e of C_i (x_ij := e_ij) => e_i Default : e_def⟩ρ
    = let  e' = C⟨e⟩ρ
           e'_i = C⟨e_i⟩ρ[x_ij]
           e'_def = C⟨e⟩ρ_def
      in   if ρ(e) "matches" C_i then e'_i
           else e_def

C⟨if b e_1 e_2⟩ρ
    = if C⟨b⟩ρ
      then C⟨e_1⟩ρ
      else C⟨e_2⟩ρ

C⟨let x_i = e_i in e⟩ρ
    = let e'_i = C⟨e_i⟩ρ[x_i]
          e''_i = reduce (x_i = e'_i)
          e' = C⟨e⟩ρ[x_i]
      in  { e''_i; e' };

C⟨simplelet x_i = e_i e⟩ρs
    = let x_i' = C⟨e_i⟩ρ[x_i]
          x''_i = new x'_i
          e' = C⟨e⟩ρ[x_i'']
      in  { x''_i; e' };

C⟨λx_i.e⟩ρ
    = let x'_i = C⟨x_i⟩ρ
          e' = C⟨e⟩ρ
          n = freshname
      in  Class n
          {     ρ(e) ENTER (x'_i : ρ(x'_i))
                {   e';
```

```
                }
            }

C⟨new e⟩ρ = new C⟨v⟩ρ

C⟨e[[e_i]]⟩ρ
    = let e' = C⟨e⟩ρ
          [[e_i]]' = C⟨[[e_i]]⟩ρ
      in  e'.ENTER^n([[e_i]]')

C⟨Closure n ae ns ael pend ttl⟩ρ
    = // Refer to the CLO heap object.
```

Listing 8.1: Translation from Mondrian to $C^\sharp$

For the most part, the translation should be fairly intuitive. We have two cases for translating variable bindings. We special case the translation for variable bindings when we are binding a lambda so that we do not build two Class declarations (one for the lambda and one for the variable binding).

The "matches" predicate in the rule for `switch` depends on the type of the switch expression. If we are matching against values of sum type, then we use the $C^\sharp$ `instanceof` function to check the runtime tag of the expression. In the case of a value type, we use a direct equivalence comparison. We describe the function call $\texttt{ENTER}^n$ in more detail in section 8.2.2. The compilation of a `Closure` has been discussed in section 4.2.2.

### *8.1.1 Functions as First Class Objects*

We have to consider the following:

1. When a delegate is generated
2. How a deligate is consistency is maintained when compiling across module boundaries
3. How many delegates are required to represent the set of all functions.

We have already considered the last point in section 7.6.1. We will look at the first two points in this discussion.

Intuitively, we create a delegate if we are applying a lambda expression to a function and when we are returning a variable that is free. We can modify the interpretation function $\mathcal{C}$ to reflect this, given in Definition 8.2.

**Definition 8.2.** *Translation scheme for pass-by-delegate.*

$\mathcal{C}\langle\lambda\overline{x_i}.e\rangle\rho$

```
     = let x'_i = C⟨x_i⟩ρ
           e'   = C⟨e⟩ρ[x_i]
           e''  = if (C⟨e⟩ρ ∈ V) then new U(ρ(e')) else e'
           n    = freshname
       in  Class n
           {       ρ(e) ENTER (x'_i : ρ(x'_i))
                   {    e'';
                   }
           }
```

$\mathcal{C}\langle e[\overline{e_i}]\rangle\rho$

```
     = let e'   = C⟨e⟩ρ
           e_i' = C⟨e_i⟩ρ
           e_i'' = forall i, if (e'_i ≡ (λx.e)) then new U(ρ(e'_i))
       in  e'.ENTER^n(e_i'')
```

Consider the following:

```
f1 : (Integer, Integer, Integer) -> Integer -> Integer;
f1 = f -> a -> f a;

f2 : (Integer, Integer, Integer) -> Integer -> Integer;
f2 = f -> x -> let g = f x;
               in  g;

f3 : ((Integer, Integer), Integer, Integer) -> Integer -> Integer;
f3 = f -> x -> f (x -> x + 1) 2;
```

In this example, `f1` takes a function `f` and applies it to `a`, returning a function. We do not generate a delegate for this because it is an application. In `f2`, `g` is not free, and we generate a delegate when we return it. In `f3`, we are passing a lambda term `(x -> x + 1)` to the function `f`. We recall that the lambda is compiled as a FUN object, which is a class in $C^\sharp$, so a delegate must be generated to pass it.

### *8.1.2 Module Compilation*

The location of a delegate definition is important. Ideally, all delegates should be defined within package `package.lang`, which means their definitions are accessible from any package since we import `package.lang` implicitly from every module. If

we did not erase types when compiling to subtype polymorphism, this would be infeasible. We would have too many possible type combinations to pregenerate.

One could argue that with a unique naming convention supplied by $\mho$, we could delay the generation of a delegate until $\mho(\sigma) \notin \mathcal{D}$. However, this complicates the compilation process, because if we generate a new delegate definition, adding it to `package.lang` would require the redistribution of the standard runtime library with every distribution of a Mondrian program.

By contrast, adding the definition to the local package interface file would require it to be statically linked whenever we used its exported functions because we do not want to regenerate the delegate definition. Doing so could lead to multiple definitions between package dependencies.

The Mondrian compiler keeps an internal table of delegate function types initialized from the package `package.lang`, given as the set $\mathcal{D}$. The restriction of $\mathcal{D}$ to $N$ is given by $\mathcal{D}/N$. We give the pregenerated functions for $\text{CLR}_{\leq}$ in Listing 8.2 and $\text{CLR}_{\leq,\forall}$ in Listing 8.3 for $N = 3$.

```
Object -> Object ["D_1"]
Object -> Object -> Object ["D_2"]
Object -> Object -> Object -> Object ["D_3"]
```

Listing 8.2: Pregenerated delegates for $\text{CLR}_{\leq}$

```
forall A => A -> A ["D_1"]
forall A , B => A -> B ["D_2"]
forall A , B , C => A -> B -> C ["D_3"]
```

Listing 8.3: Pregenerated delegates for $\text{CLR}_{\leq,\forall}$

We generate a number $N$ of delegates where $N$ is some large number, restricting function arguments to $N$.

## *8.2 Compiling $M_{CM}$ to the $\text{CLR}_{\leq}$*

Next we look at compilation of the following expressions:

1. Top-level Functions
2. The Function Call in $C^\sharp$
3. If/Switch
4. Let Bindings

### *8.2.1 Top-level Functions*

All top-level definitions are compiled in the following way in $C^\sharp$

```
class f
{
    D_2 ENTER (Integer)
    {   ....;
    }

    class f2
    {   ...
    };
};
```

Listing 8.4: Top-level $C^\sharp$ definition

where the *o*-type of `f` is given as $Integer \rightarrow Integer$. We can define `f` as the following:

```
f : Integer -> Integer;
f = (x -> let f2 = y -> ....z....;
          in  f2 x) 2 3 4;
```

The lambda in the application is typed as

$Integer \rightarrow \{Integer, \{Integer, Integer, Integer\}\}$

This is a constant applicative form or CAF. We can evaluate the expression `f 2` because the values of x and y are bound to 2 and 3. After lambda doping, we generate the following:

```
f = Closure "f"
      ((x -> let f2 = y -> ....z....; in  f2 x) 2 3) {4} 1 1);
```

The encapsulated expression can be evaluated once and stored within the top-level class definition. This gives us:

```
class f
{   public static f f_ = new fClosure();

    D_1 ENTER (Integer)
    {
        ...
    }
```

```
    class fClosure
    {   D_2 buffer = new f().ENTER(2)(3);
        Integer a  = 4;

        Integer ENTER (Integer)
        {
            ....
        }
    };
}
```

When compiling references to top-level functions, in this case `f`, we use the public static field `f_`, giving us access to the Closure instance. Thus, when `ENTER`'ing a top-level function, we are essentially entering the Closure. We note that if we were passing arguments that are not literals, then we would pass them via the Closure's constructor.

### *8.2.2 The Function Call in $C^\sharp$*

We showed in section 5.1.1 that *o*-types give us a way of statically annotating the reduction semantics. As such, they give us a convenient way of compiling a function call to the CLR. We have already encountered a few examples in which we have given the equivalent $C^\sharp$ function call for a reduction using the operational semantics.

Function application is complicated by the fact that we may be either invoking a method bound to a class instance or one bound to a delegate. In the former, we invoke the function by using the `ENTER()` method. In the latter, we invoke the method by using the `()` syntax. If the function symbol is a free variable or call continuation, we will use the `()` syntax. Otherwise, we use `ENTER`.

The type erasure algorithm given in section 7.6.5 raises a small issue when evaluating call continuations. When a call continuation is returned, which is a delegate typed as *Object* on $\text{CLR}_{\leq,\forall}$, we have to downcast the value to the appropriate delegate type.

We can summarise the $C^\sharp$ reduction rules in Table 8.1. These rules can be interpreted in the following way: given the expression `f 2 3 4` with principle function symbol `f` and the *o*-type $Integer \rightarrow \{Integer, Integer\}$, we match on rule one. The return *o*-type $\{Integer, Integer\}$ is used to match rule three. Finally, we match on rule four. The function call finally compiles to:

| Rule | $o$-type | Function Symbol | $C^\sharp$ | Return $o$-type |
|---|---|---|---|---|
| 1 | $Obj_1 \rightarrow \ldots Obj_n$ | v<br>`D_(n-1)` | `v.ENTER(...)`<br>`(D_(n-1))(...)` | $Obj_n$<br>$Obj_n$ |
| 2 | $Obj \rightarrow \{Obj_1, \ldots Obj_n\}$ | v | `v.ENTER(...)` | $\{Obj_1, \ldots, Obj_n\}$ |
| 3 | $\{Obj_1, \ldots, Obj_n\}$ | `D_(n-1)` | `(D_(n-1))(...)` | $Obj_n$ |
| 4 | $Obj$ | - | - | - |

Table 8.1: Function Call in MSM on $\text{CLR}_{\leq}$, where $Obj$ represents $Object$

```
f.ENTER(2)(3, 4);
```

### *8.2.3 If/Switch*

The conditional expressions `switch` and `if` are trivially compiled to $C^\sharp$. In the case of reference types, we use the $C^\sharp$ function `instanceof`, which performs a runtime type check on the tag of the scrutinee. $C^\sharp$ also offers a switch expression, however, it is unsuitable because it only allows switching on $Integer$ types. Therefore, we decompose the Mondrian `switch` to a series of tests using $C^\sharp$ if/then branches. In the case of value types, we can perform a simple equality comparison, using the equality operator.

### *8.2.4 Let/Simplelet Bindings*

A `let` expression creates a new local environment which consists of some $n$ number of expressions which are substituted into the let expression.

We can build the local environment in two ways: by statically allocating all expressions when the program runs, or by dynamically building the environment when we evaluate the let expression. The former has the advantage that we are not constantly reallocating and discarding if we reevaluate the expression. It does mean, however, that we may allocate an environment which we may never use. This could be solved by using a flow analysis of the style presented in (Faxn 1996*b*) or usage analysis (Wansborough & Peyton Jones 2000) which we save for future work. Currently, we will allocate the local environment when we enter the let expression.

Again, we take care to ensure eager semantics and evaluate the rhs of the

expressions with the lhs in context in the case of a `let` expression. In the case of a `simplelet`, we can simplify the compilation scheme and inline expressions by substitution into the body, although we run the risk of increasing code bloat if the expression is used in multiple locations. We can use a simple variable count to determine if the expression is used multiple times. This works for `simplelet` because we will not meet any of the following situations:

```
f = let g = x -> ...<big computation>...;
        h = g;
    in  ...h...h...
```

More aggressive inlining schemes for recursive environments could be used in the same vein as that presented in (Jones & Marlow 2002). The following `simplelet` example

```
f = x -> let // s1 : Integer -> Integer -> [Integer];
             s1 = x -> y -> map (+ 1) (x :: y);
             // s2 : Integer;
             s2 = ...x ...
         in  ...(s1 2 3)...s2;
```

is compiled to pseudo $C^\sharp$ as such:

```
class f
{
    Object ENTER (Object x)
    {   Integer s2 = ...x..;
        s1 s1      = new s1;

        ....map (+ 1) (2 :: y)... s2...
    }
};
```

The expression bound to the var `s1` has created a new class instance because it is a lambda expression and is compiled to a FUN object.

For let expressions, we either compile to a FUN or CLO heap object, depending if the expression has free environment. In essence, because FUN and CLO compile to basically the same object, that is, a class declaration, it matters little which heap object is allocated. We choose a CLO heap object for consistency.

We used the function `reduce` in the translation function $\mathcal{C}$ to reduce the let environment. `reduce` ensures that we can evaluate the rhs expressions in the context of the lhs, by performing the following:

- create a CLO heap instance for each rhs expression
- for every free environment variable in every CLO object, set its reference to the appropriate CLO instance
- call `Initialize` on each CLO instance, which will reduce the CLO expression to WHNF, if appropriate, to ensure call-by-value semantics.

For example, we can compile the following simple expression, assuming for simplicity's sake that all arguments are typed as `Object`:

```
f = x -> let // f1 : Object;
             f1 = (x -> ...f2...) 2;

             // f2 : Object;
             f2 = ...f3...;

             // f3 : Object -> Object;
             f3 = x -> ....f1..;
         in  f3 x;
```

After lambda lifting and lambda doping, we get

```
f = x -> let f1 = (Closure "f1" ((x -> ...f2....) 2) {f2} 0 0)
             f2 = (Closure "f2" (...f3...) {f3} 0 0);
             f3 = (Closure "f3" (x -> ...) {f1} 1 1);
         in  f3 x;
```

compiled to the following in pseudo $C^\sharp$:

```
class f
{   class f1 { f2 f2_; void Initialize(){...}
                       Object ENTER (...)
                       {...f2_.ENTER(...)...}
             };
    class f2 { f3 f3_; void Initialize(){...}
                       Object ENTER (...)
                       {...f3_.ENTER(...)...}
             };
    class f3 { f1 f1_; Object ENTER (Object x)
                       {...f1_.ENTER(...)...}};

    Object ENTER (Object x)
    {   f1 f1_ = new f1();
```

```
        f2 f2_ = new f2();
        f3 f3_ = new f3();

        f1.f2_ = f2_;
        f2.f3_ = f3_;
        f3.f1_ = f1_;

        f3_.ENTER(x);
    }
};
```

We recall that our discussion in section 4.2.2 details the compilation of Closures to CLO heap objects. We reduced an expression eagerly by putting it in the Closure's constructor. This, unfortunately, will not work because it is necessary to initialize all the free environment before we can reduce the right hand sides. Therefore, we compile Closures a little differently, and move the code that would have resided in the constructor to a special method called `Initialize`, which is called explicitly, once the free environment is allocated.

### *8.2.5 Lambda Erasure*

We described that lambda lifting creates a new Closure to encapsulate free environment when lifting lambdas. We can perform an easy optimisation by performing a lambda erasure, effectively removing an extra level of indirection. We also mentioned, when encapsulating a lambda expression, that we generate a unique name source where the first $n$ names are those supplied by the lambda's $n$ arguments. This makes lambda erasure trivial when compiling to a CLO heap object, as the `ENTER`'s arguments will be named as those given to the lambda's arguments. For example, we can erase

```
f = let f1 = Closure "f1" (x -> y -> ..x..y..) {...} {f1, f2} 2 2;
        f2 = Closure "f2" (z -> ..z..) {...} {f1, f2} 1 1;
    in  ..f1 ... f2..
```

to the following:

```
f = let f1 = Closure "f1" (..x..y..) {x, y,...} {f1, f2} 2 2;
        f2 = Closure "f2" (..z..) {z,...} {f1, f2} 1 1;
    in ..f1...f2 ...
```

## 8.3 Compiling $M_{CM}$ to $CLR_{\leq,\forall}$

This section is a description of the translation of the Mondrian term language to $C^\sharp$ on $\text{CLR}_{\leq,\forall}$.

### 8.3.1 The Road to Generic Compilation

There are several simple transformations we perform before compiling top-level functions. These are:

- Pretty print quantifiers for all type schemes, which amounts to generating an alphanumeric character for each quantifier
- Variables bound to polymorphic expressions, which are "quantified" using the appropriate type instantiations, discussed in section 8.3.1
- Free environment encapsulated in Closures, which are bound to polymorphic expressions, removed from the Closure's free environment list.

We take a closer look at the second and third transformations, since the first is trivial.

#### *Quantifying Variables*

We recall that in the $\text{CLR}_{\leq,\forall}$ we must instantiate a value of polymorphic type before we can use it. This is discussed in section 7.7.1. For example, the variable `id` in the expression `id 2` must be instantiated before its use. We describe here a method of quantifying a variable with the appropriate generic annotations.

Using the identity function as a simple example, its definition type is given as:

$\forall\alpha.\alpha \rightarrow \alpha$

Its instantiated type in the expression `id 2` is given as:

$Integer \rightarrow Integer \rightarrow Integer$

We can recover the type instantiation of $\alpha$ by unifying the two types. This gives us a substitution $S$ with $\{\alpha \mapsto Integer\}$. We can now instantiate `id` as the following:

```
new id<Integer>()
```

We can easily extend this to instantiating generic delegates in the same way. From this example, we can see that we require three pieces of information: the inferred type for the function definition, the instantiated type and the function symbol to instantiate. We define the function `inVar`, which generates an instantiated variable, given its definition and instantiation types. This function

`inVar(id,` $\forall\alpha.\alpha \to \alpha$ `,` $Integer \to Integer$`)`

returns the instantiated variable `id<Integer>`. We can simplify notation and assume that the definition type is always available, therefore, we only require the variable and instantiation type. If the supplied type definition is not polymorphic, then it returns the function symbol.

*Closures without Free Environment*

We remove free environment of polymorphic type from Closures because we have to instantiate an instance of a free variable at every place it is used with its instantiated type.

### *8.3.2 Top-level Functions*

Top-level functions are compiled to class declarations, discussed in section 8.2.1. This time, they are annotated with the appropriate quantifiers. For example, given the function `foldl`, typed as

$\forall\alpha\beta.(\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to [\beta]$

this is defined in $C^\sharp$ as:

```
class foldl <A, B>
{   // public static foldl<A,B> foldl_ = new<A,B> foldl();

    List<B> ENTER(D_2<A, B, B> f, A a, List<A>)
    {
        ....
    }
}
```

However we cannot generate the static instance member `foldl_`, which appears commented, because generic class instances cannot be instantiated until the types are supplied.

| Rule | $o$-type $\kappa$ | Function Symbol | $C^\sharp$ | Return $o$-type |
|---|---|---|---|---|
| 1 | $\alpha_1 \rightarrow \ldots \alpha_n$ | v<br><br>(D_(n-1)) | inVar(v, $\alpha_1 \rightarrow \ldots \alpha_n$)<br>.ENTER(...)<br>inVar(D_(n-1), $\kappa$)(...) | $\alpha_n$<br><br>$\alpha_n$ |
| 2 | $\alpha \rightarrow \{\beta_1, \ldots \beta_n\}$ | v | inVar(v, $\kappa$).ENTER(...) | $\{\beta_1, \ldots, \beta_n\}$ |
| 3 | $\{\alpha_1, \ldots, \alpha_n\}$ | D_(n-1) | (D_(n-1))(...) | $\alpha_n$ |
| 4 | $\alpha$ | - | - | - |

Table 8.2: Function Call in MSM on $\text{CLR}_{\leq,\forall}$

### *8.3.3 The Function Call*

Reduction on $\text{CLR}_{\leq,\forall}$ is, for the most part, consistent as that on the $\text{CLR}_{\leq}$, whose rules we defined in section 8.2.2. The only difference is that we have to quantify polymorphic variables and delegates with the appropriate instantiated type using the function `inVar`, which we defined earlier. The rules given in Table 8.2 can be interpreted in the same way.

### *8.3.4 If/Switch*

We compile `if` and `switch` expressions in the same way, described in section 8.2.3.

### *8.3.5 Let/Simplelet Bindings*

`let` and `simplelet` bindings compile in a similar way as that discussed in section 8.2.4. Compiling let environment by nesting is also advantageous because free generic type variables are also in context. For example, given the following, in which the variable `x` is free in the body of the function `f1`, we have:

```
// forall a => a -> b
func1 = x -> let // f1 : forall b => [b] -> b;
                 f1 = b -> ..x...
             in  f1 (...)
```

We can translate this into a class declaration in $C^\sharp$:

```
class func1 <A>
{   B ENTER (A x)
    {   ....
```

```
    }

    class f1
    {   A x;

        B ENTER (List<B> b);
        {   // uses the type variable A defined in the parent
            // class.
        };
    };
};
```

The universal quantifiers $a$ and $b$ are translated directly to the generic type parameters `A` and `B` in the class `foldr`. The appropriate type parameters are explicitly applied when an instance of a class `foldr` is used, using the function `inVar`.

We discussed in section 8.2.4 that free environment is initialised before we eagerly reduce the right hand side, and that polymorphic free environment is removed from Closures. We also remove non-polymorphic free environment because the CLI does not allow assigning values to public static fields of a generic class without providing the appropriate class instantiations. Obviously, we cannot create instances of every use of the let expression in `e`. Doing so would mean initializing the environment for every instance–an inefficient exercise. Instead, we can generate a parent class that is non-generic and add the static environment fields to the parent. Consider the following:

```
f  = x ->
   let
   { // f1 : Integer -> Integer;
     f1  = ....f2....;

     // f2 : forall a, b => a -> b;
     f2  = ..f3......;

     // f3 : forall a, b => Integer -> (a -> b) -> b;
     f3  = ..........;

     // f4 : Integer;
     f4  = ...f3.....;
   }
```

```
    in f1 x;
```

`f1` references `f2`, yet `f2` is a polymorphic class, so it will not store a environment reference. `f2` references f3, which is not polymorphic, so a static instance is created in a new parent class for `f2`. We generate the following:

```
class f1 { ... };

class f2<A, B>
{
    B ENTER (A)
    {   ...
        new f3<A, B>.ENTER( ...) ;
    }
};

class P_f3
{   f4 f4;
};

class f3<A, B> : P_f3
{
    B ENTER (A a, B b)
    {   ...
        f4.ENTER(...);
    }
};
```

The only free environment stored is that for `f4`, because `f3` and `f2` are generic. During the initialization of the `let` environment, we create an instance of `f2` as usual, and store a reference to it by accessing the parent type using `P_f3.f4 = f4`.

Unfortunately, polymorphic simplelet expressions need to be instantiated for every instance of their use, just like polymorphic recursive expressions. This means they need to be compiled inside a class definition with the appropriate generic parameters. This is similar to the process outlined above with generic recursive definitions, except it is never necessary to create a parent class.

# Chapter IX

# Conclusions and Future Work

## *9.1 Conclusions*

We have successfully implemented a transforming Mondrian compiler for a subset of the Mondrian language, $M_{CM}$, which reduces statically typed terms without runtime argument checking.

In chapter 3, we defined two reduction machines–the Mondrian Exceptional Machine (MEM), and the Mondrian System Machine ($s$-MSM). We gave the operational semantics for the MVM in chapter 4. It uses runtime argument checking to build the correct reduction path for any expression. We also presented the reduction rules for the $s$-MSM. As a result of removing runtime argument checking, they are highly simplified.

In chapter 5, we discussed how we can model the operational semantics of reduction by using annotations to the types inferred for the terms. We called these operational types, or $o$-types, because they gave us a faithful abstract model of reduction.

With an abstract model of reduction, it became possible to identify all situations where both *known* and *unknown* partial applications may occur. We then isolated these, using lambda "dopes" and saturating function application so that we removed the need for runtime argument checks.

In chapter 7, we developed a type system that infers types statically and is typeable by $CLR_{\leq}$ and $CLR_{\leq,\forall}$. We extended the $M_{CM}$ grammar so that we could annotate terms with types and parameterise class declarations. The type system is restricted in the sense that we are only inferring widening coercions over non function types. This was a result of the inability of the $CLR_{\leq}$ and $CLR_{\leq,\forall}$ to statically support typing subsumption over function types, since they do not observe the usual contra/covariance. This limited the power of the type system, because we cannot correctly infer widening coercions over function types. We showed how we can map the inferred types to the different runtime targets and the pros and

cons of each process. We presented a set of type rules for inferring terms over the $M_{CM}$ calculus. Finally, we derived a set of syntax directed decision procedures for subtyping, unification and inference, based on the type rules. This demonstrates how *o*-types could be easily generated from the substitutions generated by inference.

In chapter 8, we described a compilation scheme for translating saturated $M_{CM}$ programs to the intermediate language $C^\sharp$. We described methods for treating functions as first class values, generating delegates conservatively. All expressions are reduced eagerly, and we show how generics slightly complicates this translation by requiring that all values of polymorphic type be instantiated with their respective types.

## *9.2 Future Work*

Our discussion of further work is divided into the three core issues we identified at the start of this discussion:

1. Implementing lazy evaluation on the CLI
2. Implementing lambda doping and its subsequent performance evaluation in a commercial compiler
3. Extending the type system.

## *9.3 Implementing Lazy Evaluation*

We have yet to tackle this problem. Unlike partial applications, there are no "obvious" solutions using a program level transformation. We discussed a possible solution in the introduction, but it remains to be implemented and analysed.

## *9.4 Lambda Doping in GHC*

We have discussed the lambda doping transformation in some detail and given a translation for compiling to the *s*-MSM.

Lambda doping in its current form relies on the following:

1. An algorithm to generate *o*-types
2. PAP lifting to lift known partial applications

3. Doping to lift unknown partial applications.

GHC already posesses a type inference algorithm that is based on (Harper et al. 1987) and uses unification to resolve constraints over equalities. The doping transformation is well suited to GHC's separate compilation paradigm. Observational types can be exported in the GHC interface definition, in much the same way that the present type information is already exported. We are only annotating existing ML types, rather than generating a completely new set of type information. GHC is a optimising transforming compiler, and lambda doping would be one of many different transformation phases. We gave a number of examples in section 6.4, where we showed the interaction of lambda doping with full laziness and strictness transformations. We showed that lambda doping interacts favourably with other phases, and that there was little adverse interaction. We foresee little trouble in applying the same intuition to the remaining transformations, including deforestation and usage analysis.

With *o*-types, we can use lambda doping to remove the requirements of runtime argument checking for both a push/enter and eval/apply method of evaluation. It is interesting to note that in (Marlow & Jones 2004), Marlow & Jones gives runtime results for implementations of eval/apply and push/enter, demonstrating that eval/apply edges push/enter slightly in performance. It would be interesting to see if push/enter would regained a performance advantage in GHC. Recall that we showed that push/enter requires, at most, the same number of dopes as eval/apply, and usually less.

We could not present any meaningful performance evaluation between push/enter and eval/apply. This is because an implementation of push/enter, without using runtime argument checking, still requires an abstracted system stack in CLI. We recall that push/enter directly manipulates the stack by popping and pushing arguments. Any comparison between the two evaluation models would therefore be meaningless.

With a little thought, we can describe some immediate changes to the eval/apply model in GHC. GHC currently generates a number of common entry points which are used by the functions' slow entry point. These are called `stgApplyN` and one is generated for each value of `N`, where `N` is the arity of the function. A slow function call consists of a switch on the type of heap object and the arity of the application. Now that all function applications are known, we remove the switch and simply build direct function calls with the "known" number of arguments.

## *9.5 Type System*

We have discussed at length how we have developed a type inference algorithm to type $M_{CM}$ terms with *o*-types. We noted a number of restrictions in this algorithm, especially the lack of subsumption over function types. An obvious improvement would be the addition of a subsumption rule for function types. This would require extending the runtime target to $CLR_{\leqslant,\forall}$, which should offer no additional problems, as it is strictly more expressive then either $CLR_{\leq}$ or $CLR_{\leq,\forall}$.

Even more "adventurous" would be to restrict the type system in the spirit of (Pottier 2001). Pottier uses a type system based on constraints, generated over inequalities in a complete lattice. Unfortunately, solutions to the optimisation problems he poses assume that there is a lub and glb for every pair of types. This situation does not exist in the CLI, as there is no glb for every type. More recently, Coquery & Fages in (Coquery & Fages 2003) has laid down theoretical work in resolving constraints in a semi-lattice. This is more suited towards the CLI. However, there is still a large amount of work left in adapting Pottier's simplification algorithms in the system proposed by Coquery & Fages. Yet, despite such simplifications, constraint sets are still large and unwieldy and it is at the present unclear if a trivial translation to expressing these on the CLI exists.

As a final note, we observe that the subtype relation is not recursive (as defined in some implementations). We can model the subtype relation over finite trees, though a proof of this is not given here. Not permitting recursive types is a limitation of the $M_{\leq,\forall}$ type system. The $CLR_{\leqslant,\forall}$ supports recursive subtyping through its implementation of F-bounded polymorphism. We leave this as a possible extension.

### *9.5.1 Soundness of the Type System*

We have yet to prove the correctness of the lambda doping transformation, or the soundness of the type inference algorithm; with respect to the type rules.

We can show the soundness of the type system by using the property of subject reduction (Curry & Feys 1958), which basically states that well-typed programs "do not go wrong". The subject reduction property ensures that reductions preserve the type of expressions. Wansbrough & Peyton Jones in (Wansbrough & Peyton Jones 1999) use a similar technique to assess the validity of the usage analysis transformation. They design a type system called *UsageSP*, which infers usage type annotations for terms. Usage analysis exploits that in lazy reduction

systems; some expressions, represented by a *thunk*, are only ever used once. Updating thunks involves a great deal of expensive memory traffic. The thunk has to be allocated in the heap, re-loaded into registers when it is evaluated, and overwritten (with its value) when computed. Usage analysis allows the program to perform a number of transformations, including update avoidance and floating in. Update avoidance is similar to lambda doping in that we are altering the operational behaviour of the program. Obviously, it is important that the transformation is correct, otherwise, evaluation may result in a *stuck* expression.

In addition to subject reduction, we must also show that programs that contain type errors are not typeable. Obviously, a well-typed program should be able to reduce to a value, without violating subject reduction. However, it is possible for a program to preserve subject reduction, but still produce a type error. Consider `(/ 1 0)`. There are a number of ways for proving safety, from domain models (Damas 1984) to operational syntactic models (Wright & Felleisen 1994) and transition models over a coalgebra (K. 1996). They are compared in (Wright & Felleisen 1994). We can use the method originally inspired by (Wright & Felleisen 1994) and modify it by taking the reduction principles of bisimulation into account.

Subject reduction shows that the stability of the typing relation is preserved by the operational semantics. We can extend this by showing that the path of reduction is also consistent between lambda doped and non-lambda doped terms. This is important if we wish to establish the correctness of the doping transformation.

*Proof Sketch.* We use the property of congruence for a strongly bisimular reduction path to show that call-by-value and call-by-need semantics are preserved. For every reduction path that amounts to a partial application, there is a dope which reduces in the same fashion. □

We expect this proof should, in the most part, be routine, but it remains to be completed.

# Bibliography


Aiken, A. & Murphy, B. R. (1991), Implementing regular tree expressions, *in* 'Proceedings of the 5th ACM conference on Functional programming languages and computer architecture', Springer-Verlag New York, Inc., pp. 427–447.

Aiken, A. & Wimmers, E. L. (1992), Solving systems of set constraints, *in* 'Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science', IEEE Computer Society Press, pp. 329–340.

Aiken, A. & Wimmers, E. L. (1993), Type inclusion constraints and type inference, *in* 'Proceedings of the Conference on Functional Programming Languages and Computer Architecture', ACM Press, pp. 31–41.

Appel, A. W. (1994), 'Loop headers in lambda-calculus or cps', *Lisp and Symbolic Computation 7* pp. 337–343.

Appel, A. W. & Trevor, J. (1997), 'Shrinking lambda expressions in linear time', *Journal of Functional Programming* **7**(2), 515–540.

Augustsson, L. (1984), A compiler for lazy ML, *in* 'LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming', ACM Press, New York, NY, USA, pp. 218–227.

Baker-Finch, C., Glynn, K. & Peyton Jones, S. (2004), 'Constructed product result analysis for haskell', *Journal of Functional Programming* **14**(2), 211–245.

Boquist, U. (1999), Code Optimization Techniques for Lazy Functional Languages, PhD thesis, Chalmers University of Technology, Sweeden.

Cardelli, L. & Mitchell, J. C. (1994), Operations on records, *in* C. A. Gunter & J. C. Mitchell, eds, 'Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design', MIT Press, Cambridge, MA, pp. 295–350.
**URL:** *citeseer.ist.psu.edu/cardelli91operations.html*

Coquery, E. & Fages, F. (2003), Subtyping Constraints in Quasi-lattices, Technical report, INRIA.

Curry, H. B. & Feys, R. (1958), *Combinatory Logic, Volume I*, North Holland Publishing Company.

Damas, L. M. M. (1984), Type Assignment in Programming Languages, PhD thesis, University of Edinburgh.

Damas, L. & Milner, R. (1982), Principle type schemes for functional programs, *in* '9-th ACM Smyposium on Principles of Programming Languages', ACM Press, pp. 207–212.

Danvy, O. & Schultz, U. P. (1997), Lambda-dropping: transforming recursive equations into programs with block structure, *in* 'Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation', ACM Press, pp. 90–106.

Fairbairn, J. & Fradet, P. (1987), Tim - a simple lazy abstract machine to execute supercombinators, *in* G. Kahn, ed., 'Proc. IFIP conference on Functional Programming Languages and Computer Architecture', Springer Verlang LNCS 274, pp. 34–45.

Faxn, K. F. (1996*a*), Flow inference, code generation and garbage collection for lazy functional languages, Master's thesis, Royal Institute of Technology.

Faxn, K. F. (1996*b*), Polyvariance, polymorphism, and flow analysis, *in* '5th LOMAPS Workshop'.

Fuh, Y.-C. & Mishra, P. (1988), Type inference with subtypes, *in* 'Proceedings of the Second European Symposium on Programming', North-Holland Publishing Co., pp. 155–175.

Fuh, Y.-C. & Mishra, P. (1989), Polymorphic subtype inference: Closing the theory-practice gap, *in* 'Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2', Springer-Verlag, pp. 167–183.

Gill, A. J., Peyton Jones, S. & Launchbury, J. (1993), A short cut to deforestation, *in* 'Conference on Functional Programming Languages and Computer Architecture', ACM Press, pp. 223–232.

Girard, J.-Y., Lafont, Y. & Taylor, P. (1989), *Proofs and Types*, Vol. 7, Cambridge University Press.

Harper, R., Milner, R. & Tofte, M. (1987), A type discipline for program modules, *in* 'TAPSOFT '87', Berlin. Springer LNCS 250.

Hoang, M. & Mitchell, J. (1995), Lower bounds on type inference with subtypes, *in* 'Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages', ACM Press, pp. 176–185.

Jones, S. P. (1996), Compiling haskell by program transformation: a report from the trenches, *in* 'Proc European Symposium on Programming (ESOP'96)', Springer Verlag LNCS 1058.

Jones, S. P. & Marlow, S. (2002), 'Secrets of the glasgow haskell inliner', *Journal of Functional Programming* **12**(4), 393–434.

Jones, S. P., Nordin, T. & Oliva, D. (1997), C–: A portable assembly language, *in* 'Implementation of Functional Languages. 9th International Workshop, IFL'97', pp. 1–19.

Jones, S. P., Stestoft, P. & Hughes, J. (2004), 'Demand analysis'. To be published.

Jones, S., Partain, W. & Santos, A. (1996), Let-floating: Moving bindings to give faster programs, *in* 'Proc. Int'l. Conf. on Functional Programming (ICFP '96)', Association for Computing Machinery, Inc.

K., S. (1996), About the completeness of type systems, *in* M. de Rijke, ed., 'Observational Equivalence and Logical Equivalence', ESSLLI, p. 17.
**URL:** *http://www.cs.kent.ac.uk/pubs/1996/564*

Kennedy, A., Russo, C. & Benton, N. (2003), 'Sml.net 1.1 user guide'. User guide for the SML.NET language.

Marlow, S. (1993), Update avoidance analysis by abstract interpretation, *in* 'Proceedings of the 1993 Glasgow Workshop on Functional Programming', Workshops in Computing, Springer-Verlag, Ayr, Scotland.

Marlow, S. & Jones, S. P. (1997), 'The New GHC/Hugs Runtime System'. Unpublished.

Marlow, S. & Jones, S. P. (2004), How to make a fast curry: push/enter vs eval/apply, *in* 'International Conference on Functional Programming', ACM Press, pp. 4–12.

Meijer, E. & Claessen, K. (1997), The Design and Implementation of Mondrian, *in* 'Haskell Workshop', ACM Press.
**URL:** *citeseer.ist.psu.edu/meijer97design.html*

Meijer, E., Perry, N. & Yzendoorn, A. V. (2001), Scripting .NET Using Mondrian, *in* 'ECOOP', Springer, pp. 150–164.
**URL:** *http://link.springer.de/link/service/series/0558/bibs/2072/20720150.htm*

Milner, R. (1978), 'A theory of type polymorphism in programming', *JCSS* **17**, 248–375.

Mitchell, J. C. (1984), Coercion and type inference, *in* 'Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages', ACM Press, pp. 175–185.

Mitchell, J. C. (1991), 'Type inference with simple subtypes', *Journal of Functional Programming* **1**(3), 245–285.

Nocker, E. (1993), Strictness analysis using abstract reduction, *in* 'Proceedings of the Conference on Functional Programming Languages and Computer Architecture', ACM Press, pp. 255–265.
**URL:** *http://doi.acm.org/10.1145/165180.165219*

Nordlander, J. (1998), Pragmatic subtyping in polymorphic languages, *in* 'Proceedings of the third ACM SIGPLAN international conference on Functional programming', ACM Press, pp. 216–227.

Odesky, M., Sulzmann, M. & Wehr, M. (1999), 'Type inference with constrained types', *Theory and Practice of Object Systems* **5**(1).

Perry, N. (2002*a*), 'Eliminating Runtime Argument Checking - personal communication'.

Perry, N. (2002*b*), 'An exceptional machine'. Not published.
**URL:** *http://www.mondrian-script.org*

Perry, N. & Meijer, E. (2004), Implementing functional languages on object-oriented virtual machines, *in* 'IEE Proc.-Softw.', Vol. 151.

Peyton Jones, S. (1992), 'Implementing lazy functional languages on stock hardware: The spineless tageless g-machine', *Journal of Functional Programming* **2**(2), 127–202.

Peyton Jones, S. & Launchbury, J. (1995), 'State in haskell', *Lisp and Symbolic Computation 8(4)* pp. 293–341.

Peyton Jones, S., Ramsey, N. & Reig, F. (1999), C–, a portable assembly language that supports garbage collection, *in* G. Nadathur, ed., 'International Conference on Principles and Practice of Declarative Programming', Berlin, pp. 1–28. Lecture Notes in Computer Science.

Pierce, B. C. (1994), 'Bounded quantification is undecidable', *Inf. Comput.* **112**(1), 131–165.

Pottier, F. (1998), A framework for type inference with subtyping, *in* 'Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP'98)', pp. 228–238.

Pottier, F. (2000), 'A versatile constraint-based type inference system', *Nordic Journal of Computing* **7**(4), 312–347.

Pottier, F. (2001), 'Simplifying subtyping constraints: a theory', *Information & Computation* **170**(2), 153–183.

Remy, D. (n.d.), Type inference for records in a natural extension of ML, Technical Report RR-1431, INRIA.
**URL:** *citeseer.ist.psu.edu/remy91type.html*

Serrano, M. (1995), A fresh look to inlining decision, *in* 'Proceedings of the 4th International Computer Symposium (ICS'95)', Springer-Verlag.
**URL:** *citeseer.ist.psu.edu/serrano95fresh.html*

Serrano, M. (1997), Inline expansion: When and how?, *in* 'Proceedings of the 9th International Symposium on Programming Languages: Implementations, Logics, and Programs', Springer-Verlag, pp. 143–157.

Smith, G. S. (1994), 'Principal type schemes for functional programs with overloading and subtyping', *Science of Computer Programming* **23**(2-3), 197–226.

Smith, J., Perry, N. & Meijer, E. (2002), 'Mondrian for .NET', *Dr Dobb's Journal* .
**URL:** *http://research.microsoft.com/ emeijer/Papers/MondrianDDJ.pdf*

Syme, D. (2001), 'ILX: Extending the .NET common IL for functional language interoperability', *Electronic Notes in Theoretical Computer Science* **59**(1).
**URL:** *citeseer.ist.psu.edu/syme01ilx.html*

Syme, D. (2002), 'Ilx sdk'.
**URL:** *http://research.microsoft.com/projects/ilx/ilx.aspx*

Trifonov, V. & Smith, S. (1996), Subtyping constrained types, *in* 'Proceedings of the third International Static Analysis Symposium', Vol. 1145, LNCS, pp. 349–265.

Urzyczyn, P. (1992), 'Type reconstruction in $f_\omega$ is undecidable', *TLCA [TLCA92]* pp. 418–432.

Wadler, P. (1988), Strictness analysis aids time analysis, *in* '15'th ACM Symposium on Principles of Programming Languages', ACM Press.

Wadler, P. & Hughes, R. J. M. (1987), Projections for strictness analysis, *in* '3'rd International Conference on Functional Programming Languages and Computer Architecture'.

Wand, M. & O'Keefe, P. (1989), On the complexity of type inference with coercion, *in* 'Proceedings of the fourth international conference on Functional programming languages and computer architecture', ACM Press, pp. 293–298.

Wansborough, K. & Peyton Jones, S. (2000), Simple usage polymorphism, *in* 'ACM SIGPLAN Workshop on Types in Compilation', ACM Press.
**URL:** *citeseer.ist.psu.edu/article/wansbrough00simple.html*

Wansbrough, K. & Peyton Jones, S. (1999), Once Upon a Polymorphic Type, *in* 'Twenty-sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages', ACM Press, San Antonio, Texas.

Wells, J. B. (1993), 'Typeability and type checking in the second-order lambda-calculus are equivalent and undecidable', *Tech. Rep.* .

Wright, A. K. & Felleisen, M. (1994), 'A syntactic approach to type soundness', *Information and Computation* **1**(115), 38–94.